



Technische Universität Darmstadt,
Fachbereich Informatik.
Genehmigte Dissertation zur Erlangung
des akademischen Grades Dr. rer. nat.

Persistent Arrays, Path Problems, and Context-free Languages

Dissertation – vorgelegt von
Dipl.-Inform. Oliver Glier
geboren in Frankfurt am Main

Tag der Einreichung: 31. Januar 2005

Tag der mündlichen Prüfung: 21. April 2005

Darmstädter Dissertationen D17

Referenten:

Prof. Dr. Hermann K.-G. Walter, Darmstadt

Prof. Dr. Jürgen Albert, Würzburg

Priv-Doz. Dr. Ulrike Brandt, Darmstadt

0.1 Deutschsprachige Zusammenfassung

Das Ziel dieser Arbeit ist es, einen Rahmen für die Modellierung sprachbeschränkter Wegeprobleme in Graphen zu schaffen. Von konzeptioneller Seite sind da kontextfreie Grammatiken und Semiringe zu nennen. Kontextfreie Grammatiken erlauben die Beschreibung rekursiver generativer Prozesse, wie sie beispielsweise in der Grammatik natürlicher Sprache auftauchen oder zur Beschreibung von Zellwachstum dienen können. Sie erfassen außerdem die Struktur einiger dynamischer Programmierschemata, so wie sie zum Beispiel im CYK Algorithmus für kontextfreies Parsing oder in manchen pseudopolynomiellen Algorithmen für schwere Probleme benutzt werden.

Die Verbindung kontextfreier Grammatiken mit Wegeproblemen in Graphen erlauben die Modellierung zusätzlicher formalsprachlicher Einschränkungen der möglichen Wege. Auf der anderen Seite können kontextfreie Grammatiken analog zum algebraischen Wegeproblem mit Semiringwerten ausgestattet werden. Im vierten Kapitel dieser Arbeit werden diese Beziehungen zwischen semiringbewerteten kontextfreien Grammatiken und Wegeproblemen näher untersucht. Diese Verbindung ist fruchtbar und erlaubt einen konzeptionellen Rahmen für Verallgemeinerungen des Wegeproblems für Graphen, beispielsweise zur Formulierung von Polynomialzeitalgorithmen für Transportprobleme welche sonst gar nicht oder nur umständlich als konventionelle Wegeprobleme dargestellt werden können.

Wie sich beobachten lässt, können formalsprachliche Einschränkungen dazu führen, dass die erlaubten Wege exponentiell lang werden. Persistente Arrays, welche als binäre Bäume dargestellt werden, erlauben mit diesem Problem umzugehen und erhalten die Eigenschaft der Polynomialzeitberechenbarkeit. Allerdings wirft die Darstellung bestimmter exponentiell langer Wege in polynomiell Platz die Frage auf, welche Berechnungen überhaupt auf sie angewandt werden können. Tatsächlich scheint sogar die Bestimmung, ob zwei solche Arrays äquivalent sind, schwer durchführbar zu sein. Im dritten Kapitel zeigen wir neben anderen Algorithmen für persistente Arrays einen probabilistischen Äquivalenztest. Persistente Arrays als solche werden bereits im ersten Kapitel kurz vorgestellt. Dort zeigen wir auch, dass das „Zippen“ zweier persistenter Arrays exponentieller Länge, welche als Bäume dargestellt werden, im Allgemeinen nicht in einer Zeit durchgeführt werden kann, welche polynomiell zu deren Größe im Speicher wächst.

Das zweite Kapitel führt in Wegeprobleme in Graphen ein. Der erste Abschnitt befasst sich mit Wegeproblemen in dynamischen Wäldern. Auch wenn der resultierende Algorithmus nicht so schnell wie die bestbekannten Algorithmen ist, erlaubt er eine einfache Implementierung, falls persistente Arrays bereits in einer Programmierbibliothek vorhanden sind. Wir stellen außerdem Semiringe und das algebraische Wegeproblem in Verbindung mit persistenten Arrays vor.

Kapitel 5. zeigt einige Ergebnisse aus der Theorie formaler Sprachen, welche sich während dieser Arbeit ergeben haben. In Kapitel 6. zeigen wir weiterhin, wie Rytters Formulierung von Valiants subku-

bischem Algorithmus für kontextfreies Parsing mit Semiringparsing und Wegeproblemen in Verbindung steht.

0.2 Preface

The goal of this thesis is to provide a framework for several path problems for graphs. From the conceptual part, there are context-free grammars and semirings. Context-free grammars allow us to describe recursive generative processes, such as in grammars for natural languages or in models for the growth of cell complexes. They also capture the structure of certain dynamic programming schemes, as it is used in the CYK algorithm for context-free parsing and of some pseudo-polynomial algorithms for hard problems.

The connection of context-free grammars with path problems in graphs allows us to model several language-restrictions for feasible paths and thus generalizes the classic shortest-path problem. On the other hand — similarly to the algebraic shortest path problem — context-free grammars can be equipped with semiring values. In chapter 4 of this thesis, we explore the relationship between semiring-valued context-free grammars and path problems. The connection is fruitful since it gives us a conceptual framework for many generalizations of the path problem. For instance it allows us to formulate tentative polynomial-time algorithms for transportation problems which cannot, or only with great difficulty, formulated as conventional shortest-path problems.

As one can observe, language-restrictions on paths might cause the shortest feasible paths to grow exponentially. Persistent arrays, represented as balanced binary trees, allow us to deal with this problem and still maintain polynomial-time computability. The ability to represent certain exponentially long arrays in polynomial space gives rise to the question what kind of computations they apply to. Indeed, even determining equality for two of these arrays appears to be difficult. We provide a probabilistic equality test in the third chapter, among other algorithms for persistent arrays. Persistent arrays themselves are briefly introduced in the first chapter of this thesis. There, we also show that zipping two persistent arrays (represented as trees) is generally not possible in time polynomial in its size in memory.

The second chapter sets the stage for path problems on graphs. The first section shows how shortest paths in forests can be maintained dynamically. The resulting algorithms are somewhat slower than the best algorithms known in the literature, but easier to implement if persistent arrays are already provided by a programming library. We also introduce the semiring framework for path problems here.

Chapter 5. shows some results in formal language theory which are useful for our conceptual framework. In chapter 6. we show how Rytter's formulation of Valiant's subcubic context-free parsing algorithm is connected to semiring parsing and path problems.

0.3 Acknowledgements

Credit is due to many people who inspired, encouraged, and supported me during my research and during the process of writing this thesis. First of all, I want to mention Professor Hermann Walter, who guided my research and who gave me such an inspiring working environment. Of similar importance to my work is Ulrike Brandt: Nearly all of my knowledge about the theory of formal languages I learned from her and from Professor Walter. They introduced me to Kolmogorov complexity and to Valiant Parsing which — as I think — lead to the most valuable results of this thesis. Without both of them this thesis would not exist. I'm very grateful to Professor Jürgen Albert for his expertise and his patience while going through the early drafts of my thesis. To all three of the aforementioned referees I owe numerous ideas, corrections, and critics of my work. It must have been as painful for them as it was for me to see all my mistakes which they so kindly corrected.

I also want to mention Professor Helmut Waldschmidt and Klaus Guntermann. I was temporarily staying with their research group for system programming. During this semester I was introduced into various fields of graph algorithms and much of this thesis is originated from there. I'm very glad that I was able to work at Professor Walter's research group for automata theory and formal languages: I want to thank all members of this group and of the computer science department of TU Darmstadt. Especially Elfi Steingasser, Jürgen Kilian, and Kai Renz have been truly supporting and encouraging colleagues during all this time. It is very sad to realize that this group does not exist anymore.

This work would never have been completed without the driving force of my friends Lars Bindzus and Nima Barraci. They both helped me to get my English into a readable form. The layout of this thesis is due to Lars Bindzus who made an ambitious job with a great result.

The DAAD provided me with a generous scholarship for conducting initial parts of my research at Fakultas Ilmu Komputer at Universitas Indonesia. I want to thank Dr. Chan Basaruddin and my colleagues from Fakultas Ilmu Komputer for the opportunity to work there.

Last but not least I thank Betti Said and my parents Helga and Arthur Glier for their enduring support and patience in all these years.

Contents

0.1	Deutschsprachige Zusammenfassung	i
0.2	Preface	ii
0.3	Acknowledgements	iii
1	Introduction	1
1.1	Strings and Arrays	1
1.2	Fully Persistent Arrays	4
1.3	Examples for Applications	5
1.3.1	Programming Languages	6
1.3.2	Language Restricted Path Problems	6
1.4	The ADT Persistent Array	7
1.4.1	The Interface	8
1.4.2	An Implementation of Persistent Arrays as Balanced Trees	9
1.4.3	Homomorphisms on the Free Monoid	14
1.4.4	Impossibility of Pairing	16
1.5	Related Work	19
2	Graph Algorithms and Persistent Arrays	21
2.1	Paths in Dynamic Trees and Forests	22
2.1.1	Path Queries on Static Trees	23
2.1.2	Path Queries on Dynamic Trees	25
2.1.3	Diameter, Center, and Diameter Path	29
2.2	The Semiring Framework for Shortest Paths	31
2.2.1	All-Pairs-Shortest-Distance	32
2.2.2	Semirings for Paths	33
2.3	All Pairs-Shortest-Paths and Successor Matrices	37
2.4	Related Work	39
3	Homomorphisms on Arrays	41
3.1	Algorithms on Large Arrays	41
3.1.1	Polynomial Evaluation and Inequality Tests	41

3.1.2	Lexicographic Comparison	45
3.1.3	Equality Tests for Arrays which May Contain Arrays	45
3.1.4	Sorting of Elements	46
3.1.5	Generalized State Machine and Acceptor	46
3.2	Algorithm Design	47
3.2.1	Poisson Trials	47
3.2.2	Interactive Line Breaking	52
3.2.3	Related Work	53
4	Language-Restrictions and Minimal Syntax-Trees	55
4.1	Semiring-Valued Context-Free Grammars	56
4.2	Language-Restrictions and Minimum Syntax-Tree	58
4.2.1	Reducing the CF-APSD Problem to the APMS Problem	59
4.3	Solving The All-Pairs-Minimum-Syntax-Tree Problem	61
4.3.1	Finite Languages	64
4.4	Applications	66
4.4.1	Combinatorial Path Problems	67
4.4.2	Transportation Problems	67
4.4.3	Algebraic Minimum Syntax-Tree	68
4.4.4	Deriving Pseudo-polynomial Algorithms	68
4.5	Related Work	70
5	Results from Formal Language Theory	71
5.1	Permutations of Strings	71
5.2	Associativity and Regularity	74
5.3	Kolmogorov Complexity and Deterministic Context-Freeness	77
5.3.1	A Short Introduction into Kolmogorov Complexity	77
5.3.2	The KC-DCF Lemma	79
5.4	Related Work	84
6	Valiant Parsing	87
6.1	Semiring Parsing and Transitive Matrix Closures	88
6.2	Transitive Closure via Shortest Paths	91
6.2.1	Bottom-Up Properties	91
6.2.2	Strongly Congruent Grids and Shortest Paths	93
6.3	Applications	99
6.4	Related Work	100
A	Definitions from Graph Theory	101

Chapter 1

Introduction

An indexed sequence of data elements $a[0], a[1], \dots, a[n-1]$ is often called “array” (throughout this thesis, array indices start at 0). A set of subsequent memory cells is a possible implementation of an array, which is often used in imperative programming languages. However, in this thesis an array is considered as an abstract data type. An interface for an abstract data type of dynamic arrays will be described in section 1.4. Almost every non-trivial program uses arrays of some form: Even the most basic data structures of every programming environment like strings, arrays, lists, stacks, queues, etc. can all be understood as instances of arrays. From one perspective, one can regard an array as an abstract data type, which allows to store, access, and modify the elements of a given array. From an implementation-specific perspective, arrays are a way of arranging elements in form of a data structure such that those operations can be performed efficiently.

In this introductory chapter we will make this distinction more precise. Furthermore, we will discuss so-called *persistent arrays* and their standard implementation in form of binary trees. Operations on persistent arrays are non-destructive, i.e. they leave old versions unaltered — including concatenation and splitting. This has a remarkable consequence for the representation of the underlying trees in memory: Through multiple references of inner nodes, common subarrays might be shared multiple times among arrays or within the very same array. Particularly, it is possible to represent arrays of exponential length within a polynomial amount of memory. Later in the fourth chapter — during the discussion of the language restricted path problem — we will see that this remarkable property is indeed useful. That is, there are applications where exponentially long arrays emerge naturally during the course of computation. In this chapter, we give time bounds for machines equipped with array operations for concatenation and splitting, and we also show how information can be extracted from arrays in terms of homomorphic images.

1.1 Strings and Arrays

We start with an example: A typical programming task is to map a string to another string. String processing is so important that entire programming languages have been designed with this purpose in

mind, for instance the language Perl. Almost all programming languages offer some representation of strings as a primitive type in some way, or at least as a fundamental part of their programming libraries. Consider the task of being given a string $v \in X^*$ of length n , where all symbols x_0, \dots, x_{n-1} of $v = x_0 \dots x_{n-1}$ are taken from the set $X = \{0, 1, 2, \dots, k-1\}$. Now, additionally given a family of strings $s_i \in X^*$, $i = 0..k-1$, the task is to compute the image of v under the substitution $i \mapsto s_i$, i.e., to generate the string $s_{x(1)} \dots s_{x(n)}$. The task is not difficult, but has some pitfalls if implemented naively.

Before considering any code, some words on notation first: If not stated otherwise, all **strings** are considered to be finite sequences (also called **words**). Usually, the letters X, Y, \dots denote the set of possible symbols of the strings in consideration. The set of all **strings over** X is denoted by X^* , which together with concatenation \cdot and the empty string ε forms the **free monoid** over X . We prefer to use the term “string” instead of “word”. When writing programs which operate on strings or on any other mathematically well-defined object, we must distinguish

1. the abstract data type which corresponds to the mathematical object “string” including all required operations on it, and
2. the data structures and algorithms that implement the abstract data type.

In order to make this distinction clear, we present a program of our introductory programming task:

Algorithm 1.1 simultaneous string substitution

```

v is array of 0..k-1
s is array[0..k-1] of array of 0..k-1
v := ReadLine()
n := length(v)
for i = 0..k-1 do
    s[i] := ReadLine()
end for

v' is array of 0..k-1
v' := emptyArray
for i = 0..n-1 do
    v' := v' · v[s[i]]
end for
return v'

```

For all code in this thesis, we use a pseudo programming language, which is similar to C or Pascal and which is not explained any further. Whenever necessary, some remarks will hopefully make the meaning clear: In this example, all strings are implemented as arrays of unbounded size, while the

family of strings s_i , $i = 0..k - 1$ is implemented as a fixed-size array of strings. After reading the input, the result v' is computed by iterative concatenation of all $v[s[i]]$, $i = 0..n - 1$. This implementation is straightforward, so what is wrong with it? The answer depends on the underlying representation of the data type array. In many programming languages, an array a of length n is represented by a sequence of n subsequent memory cells. In case that the array elements have a fixed size or alternatively when they are represented by pointers to their location in memory, then the i th element $a[i]$ of array a can be accessed by computing the memory address

$$\text{ADDRESS}[a, i] := \text{BASEADDRESS}(a) + i \cdot \text{ELEMENTSIZE} .$$

The drawback of storing an array in subsequent memory cells is that appending elements to the array can be costly: if the subsequent memory cells behind $\text{ADDRESS}(a, n - 1)$ are already occupied by other objects, then appending a single new element requires to copy the first n elements of a to another sufficiently large memory area. When this is done naively, then each concatenation in the program above consumes $O(n)$ time, even if the new strings s_i are negligibly short. The total running time of the algorithm above is then $O(n^2)$.

Fortunately there are many alternative techniques for implementing **extendable arrays**, as we will also discuss in appendix B. Those techniques usually lead to constant running time (which sometimes only amortized) for appending a new single element.¹ Now, if each of the s_i can be considered to have negligible length compared to v , $O(1)$ time for appending single elements leads to $O(n)$ total running time for the program above.

After all, our program depends on an efficient implementation of the abstract data type (ADT) “array” with operations for appending single elements to the right as well as accessing elements $a[i]$ by their respective index i . The example also shows the distinction between strings as mathematical objects, and their representation by a suitable ADT. It is sometimes worthwhile to restrict the interface of the ADT ARRAY: Such restrictions may permit more efficient implementations, and also ensure correct use of the array within the given context.

1. **Stacks** allow appending and removing single elements on one side.
2. **Queues** allow appending single elements on one side and removing single elements on the other side.
3. **Deque (Double Ended Queues)** allow appending and removing single elements on both sides.
4. **Lists** are likewise but do not allow direct access to elements by their index. Depending on the implementation of the lists, they have other well-known interfaces.

All four ADTs can be implemented in such a way that their operations need only constant time.

¹For example the class VECTOR of the JAVA API or the deque implementation of C++ Standard Template Library (STL)

In the following section we will see how the entire set of operations of the ADT ARRAY can be implemented in such a way that each operation is sufficiently efficient. That is, all operations — including concatenation and splitting — need time logarithmic in the length of the given arrays. The standard implementation are binary trees, which have the additional advantage of being fully persistent, i.e. operations on them do not destroy old versions of the data structure.

1.2 Fully Persistent Arrays

Let us consider the following sequence of assignments:

```
A := [1, 2, 1, 2, 1, 2]
B := A
B[2] := 3
```

The question is: What is the value of $A[2]$ in the end? Except for strictly functional programming languages, for most programming languages the answer depends on whether the assignment $B \leftarrow A$ copies the entire array $[1, 2, 1, 2, 1, 2]$ to B or B is assigned to the same address as A . In the former case $A[2]$ will remain to be of value 2, but the drawback is that the assignment $B \leftarrow A$ may take as long as copying all elements from A to B . So most imperative programming languages which stress efficiency have the latter semantics for the primitive datatype ARRAY: A and B will refer to the same array such that changing either one of them implies changing the other as well. In this case, $A[2]$ becomes 3, too. When teaching computer science at school, this difference in semantics of references compared to other primitive data types such as integers can be counter-intuitive. Many mistakes and programming errors are based on having different names for the same object.

Definition 1 *We call an ADT **fully persistent** if, additionally to its interface, all operations retain the operands unaltered, in such a way that the operands and the results of such operations can be freely reused during the further course of computation.*

Under abuse of language, we call a data structure (as the implementation of an ADT) fully persistent, if it implements all operations of the intended, fully persistent, ADT efficiently, which usually means in poly-logarithmic time or in time only poly-logarithmically worse compared to the fastest known non-persistent data structure.

Since we do not consider weaker concepts of persistency, we shortly use the term “persistent” instead of “fully persistent”.² Parts of the definition above are left intentionally vague and leave room to argue whether a particular data type or structure is persistent or not. We will, however, be more precise when it comes to particular instances of persistent data structures. Taking the concept of reuse of data to its

²Weaker concepts of persistency are interesting since they often allow a more efficient implementation and have many applications in their own right. See for instance [Oka 98]

limits, consider the following code snippet, where the ADT of A is a fully persistent array:

```

A := [1, 2]
for i = 1..n do
    A := A · A
end for

```

The array A is initialized with $[1, 2]$ and then n times concatenated with itself. In the end, A will have length 2^{n+1} . Thus, already for relatively small n , the length of A can be enormous. On the other hand, if persistent arrays are implemented by a suitable persistent data structure, the program needs only $O(n^2)$ time. Therefore only $O(n^2)$ memory cells will be occupied by A — how can such a long array be represented in so little space and be constructed in such a short time? The reason is that the final array A is very regular, and that, provided with a suitable implementation for persistent arrays, the construction of A in the for-loop generates a representation of the array where repetitive subarrays are mutually shared:

The standard implementation for persistent arrays are balanced trees, such as AVL-trees or B-trees. We give a short description of our implementation in section 1.4.2. Though AVL-trees are well-known, some details are necessary since many techniques rely on the actual representation in memory. For now, we only note that even though the logical representation of persistent arrays are balanced trees, their layout in memory is a graph which allows several nodes to share common subtrees. Thus, when we have many repetitions in arrays, or when many arrays share common subarrays, we often find that their representation in memory is much more compact in size than the sum of the array's respective lengths suggests.

1.3 Examples for Applications

There is a conceptual advantage if unrestricted use of array operations is encouraged. For instance, consider the following program for sorting an array, where $\text{HEAD}(A, p)$ yields, one after the other, the first p elements of the array A and $\text{TAIL}(A, p)$ yields the rest.

```

n := LENGTH(A)
B := ε (empty array)
for i = 0..n - 1 do
    x := A[i]
    p := BINARYSEARCHPOSITION(A, x)
    B := HEAD(B, p) · [x] · TAIL(B, p)
end for

```

Its running time is $O(n(\log n)^2)$ by using the standard implementation of persistent arrays: $O(\log n)$ array accesses during the binary search for each of the n insertion positions, and $O(\log n)$ time for each

of the array accesses and all other array operations. This is not very fast compared to $O(n \log n)$ time algorithms such as mergesort, but it might be efficient enough for many purposes, especially in programming environments that implement persistent arrays efficiently³.

Other basic applications of persistent arrays are thinkable: For instance we can use fast exponentiation in order to create an array of n consecutive zeros (or any other repetition) in $O((\log n)^2)$ time and then use the resulting array as a presentation for sparse vectors.

1.3.1 Programming Languages

Insufficient knowledge of the actual implementation of a data type often leads to poor performance and insufficient scalability of the resulting program. This is particularly true for several implementations of lists and arrays: For instance, when appending elements to a single-linked list, novice programmers often overlook the difference in asymptotic performance between appending at the list's front and appending at its end. Finally, the problem of lost referential integrity still remains in many programming languages which allow to change an object's state globally — either by a global variable or by passing a reference to the object. In most imperative and hybrid (in the sense of not purely declarative) programming languages implementations of "array" suffer from this. On the other hand arrays are abundant in most algorithms. We hope that this thesis encourages language designers to implement persistent arrays directly as a primitive data type into their language. If implemented in a low level language such as in C, and with additional programming optimizations not mentioned in this thesis, performance could still be comparable to many interpreted languages.

1.3.2 Language Restricted Path Problems

The benefit of multiple references to infix-substrings becomes more obvious with our next example. Consider the maze which is depicted in figure 1.3.2. We want to go from the house to the well by using the fastest way possible. The time needed to traverse the path consists of the time for walking with a given speed plus the time needed to earn a certain amount of coins: Each time we pass a bridge we have to pay a toll. The exact amount we have to pay is printed above each bridge. There are particular sites where we may earn coins. The working time in minutes for earning a single coin is also shown in the picture.

Assuming the maximum costs per bridge are bounded, we can separate concerns by assigning two values to each path π which leads us from the house to the well: The first value is the distance $\delta(\pi)$, which tells us how much time we need for traversing the path. The second value is a string $\phi(\pi) \in X^*$, where the characters $X = \{+, -\}$ symbolize a "+" for each collected coin and a "-" for each coin we give away when passing a bridge. Now we only have to ensure that each prefix of $\phi(\pi)$ contains at least as many + as it contains -. This property can be easily verified by some stack machine M . So what we

³Note that on most modern hardware, there are comparable hidden logarithmic costs for random access, for instance the paging tree, the memory hierarchy, etc

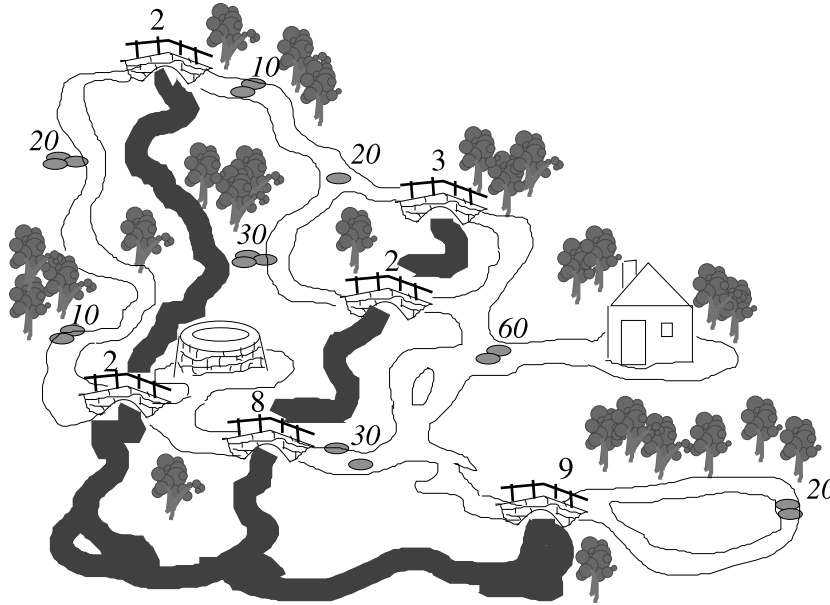


Figure 1.1: Find a way to go from the house to the well as fast as possible: The reader may assume a walking speed of 2cm/min within the scale of the picture.

have to do is to find a path π with smallest $\delta(\pi)$ such that $\phi(\pi)$ is in the language accepted by the stack machine M .

What we find here is an instance of the so-called context-free language restricted path problem, which can be solved in polynomial time. One characteristic feature of its solutions is that the paths can contain cycles: For instance, in our maze it might be worthwhile to consider to return to a location after a detour for earning coins. In general, one may even impose other context-free restrictions on $\phi(\pi)$ such that any feasible path π has exponential length (exponential in the size of the given grammar). As we will see in the fourth chapter, even in this case the feasible paths can be constructed by using only polynomially many concatenations.

What follows in this chapter is the description of the ADT PERSISTENT ARRAY and its implementation. We will also show that arrays can always be constructed in polynomial time and be represented in polynomial size if they are generated by a polynomial number of array operations.

1.4 The ADT Persistent Array

Having shown the benefits of fully persistent arrays, we will now summarize the operations on them and describe briefly their implementation as balanced trees. We also give time bounds for machines with array operations that use this implementation. Additionally we show how homomorphisms can be computed by traversing the memory representation of the array.

While on one hand this section illustrates the versatility of persistent arrays, we will demonstrate that on the other hand elementwise pairing of two equal-length arrays is not possible with our implementation.

1.4.1 The Interface

The interface of the ADT PERSISTENT ARRAY consists of the following operations:

1. $[x]$ is the singleton constructor which takes an element x and returns the array which contains x as its only element.
2. ε is the empty array, sometimes also denoted by $[]$.
3. $\text{LENGTH}[A]$ returns the length of the array A . This is also written as $|A|$.
4. $A[i]$ takes an array $A = [a_0, \dots, a_{n-1}]$ and an index $i = 0..n-1$, then returns a_i .
5. $\text{CONCATENATE}(A, B)$ concatenates two arrays: if $A = [a_0, \dots, a_{n-1}]$ and $B = [b_0, \dots, b_{m-1}]$, $n, m \geq 0$, the result will be $[a_0, \dots, a_{n-1}, b_0, \dots, b_{m-1}]$. We will often write $A \cdot B$ instead of $\text{CONCATENATE}(A, B)$.
6. $\text{HEAD}(A, i)$ takes an array $A = [a_0, \dots, a_{n-1}]$ and an arbitrary integer i , then returns the array $[a_0, \dots, a_{\min(i-1, n-1)}]$. If $i < 0$ the result is $[]$.
7. $\text{TAIL}(A, i)$ takes an array $A = [a_0, \dots, a_{n-1}]$ and an arbitrary integer i , then returns the array $[a_{\max(i, 0)}, \dots, a_{n-1}]$. If $i \geq n$ the result is $[]$.

Throughout this thesis, we will use the notational convention that for $j < i$, a_i, \dots, a_j denotes the empty sequence and that sums (any other associative operation with unit element) over empty sequences yield 0 (the unit element). Furthermore, for a finite sequence a_0, \dots, a_{n-1} and natural numbers i, j , the subsequence a_i, \dots, a_j is the sequence $a_{\max(i, 0)}, \dots, a_{\min(j, n-1)}$. With these conventions, $\text{HEAD}(A, i) \cdot \text{TAIL}(A, i) = A$ holds for all arrays A and integers i .

For convenience, we may often identify an array $[a_0, \dots, a_{n-1}]$ with the corresponding string $a_0 \cdots a_{n-1}$ of elements. We will, however, make a distinction between the underlying structure of an implementation, and if B is such a structure, then $\text{seq}(B)$ denotes the string which is represented by B .

Based on the operations above, the ADT can be extended by the following operations and macro definitions:

1. $[x_0, \dots, x_{n-1}]$ is a shorthand for

$$\text{CONCATENATE}([x_0], \text{CONCATENATE}([x_1], \text{CONCATENATE}(\dots, \text{CONCATENATE}([x_{n-2}], [x_{n-1}])))).$$

Note that whenever we write $[x_0, \dots, x_{n-1}]$ in a program, evaluation of this expression always involves $n-2$ concatenations.

2. $\text{REVERSE}[A]$ takes an array $A = [a_0, \dots, a_{n-1}]$ and returns $[a_{n-1}, \dots, a_0]$. We can implement this with a constant factor overhead by maintaining two versions of an array A : one is in the current order of elements of A , the other is in the reversed order. $\text{REVERSE}[A]$ will simply swap the two versions.
3. $\text{REPLACE}[A, i, x]$ takes an array $A = [a_0, \dots, a_{n-1}]$, an index $i = 0..n-1$, and a new element x , then returns $[a_0, \dots, a_{i-1}, x, a_{i+1}, \dots, a_{n-1}]$. This can be implemented by $\text{REPLACE}[A, i, x] = \text{HEAD}(A, i) \cdot [x] \cdot \text{TAIL}(A, i+1)$, however, since this operation is frequently used in some programs, it will usually be implemented more efficiently.
4. $A[i] \leftarrow x$ is a shorthand for the assignment $A \leftarrow \text{REPLACE}(A, i, x)$. Similarly, if $A[i]$ itself is an array, $A[i][j] \leftarrow x$ is a shorthand for the assignment $A \leftarrow \text{REPLACE}(A, i, \text{REPLACE}(A[i], j, x))$ and so on for higher dimensions.
5. $\text{INSERT}[A, i, x]$ takes an array $A = [a_0, \dots, a_{n-1}]$, an index $i = 0..n-1$, and a new element x , then returns $[a_0, \dots, a_{i-1}, x, a_i, \dots, a_{n-1}] = \text{HEAD}(A, i) \cdot [x] \cdot \text{TAIL}(A, i)$, which is an array of length $n+1$.
6. $\text{REMOVE}[A, i]$ takes an array $A = [a_0, \dots, a_{n-1}]$ and an index $i = 0..n-1$, then returns $[a_0, \dots, a_{i-1}, a_{i+1}, \dots, a_{n-1}] = \text{HEAD}(A, i) \cdot \text{TAIL}(A, i+1)$, which is an array of length $n-1$.

The usefulness of the operation REVERSE becomes apparent in the next chapter.

1.4.2 An Implementation of Persistent Arrays as Balanced Trees

We will use balanced trees as our standard implementation of persistent arrays. This is quite common. Since implementations of balanced trees like AVL-trees and B-Trees can be found in any textbook on basic algorithms and data structures, we will only sketch our implementation briefly.

A crucial point is that we allow multiple references to the same subtree: While the conceptual representation of persistent arrays are directed trees, their representation in memory is different. We allow that a node can be the successor of many different nodes. Therefore we define under abuse of the term "tree":

Definition 2 *Let X be some set. A **binary tree with multiple references** over X is tuple $B = (N, L, R)$ with $L \cap R = \emptyset$, $L, R \subseteq N \times N$, such that $(N, L \cup R)$ is a rooted acyclic directed graph with nodes N and edges $L \cup R$ and in each one of the two subgraphs (N, L) and (N, R) each node has at most one direct successor. Furthermore, the nodes without any successors (the leaves of B) must all be elements of X .*

The intention behind definition 2 is that for each node $p \in N$, its left child is (if it exists) in the set L , while its right child is (if it exists) in the set R . Note that a binary tree with multiple references is in general not a directed tree in a graph-theoretic sense because there can be more than one path from the root to one of its successors. However, our abuse of language corresponds to common practice in functional programming, where the term "with multiple references" is usually omitted.

Given a binary tree with multiple references $B = (N, L, R)$ and a node $p \in N$, we write $\text{LEFT}(p) = q$ if q is the unique successor of p in (N, L) , and if no such q exists we write $\text{LEFT}(p) = \perp$, assuming $\perp \notin N$. Similarly, we write $\text{RIGHT}(p) = q$ if q is the unique direct successor of p in (N, R) and $\text{RIGHT}(p) = \perp$ if no such q successor exists. We call $\text{LEFT}(p)$ the **left child** of p and $\text{RIGHT}(p)$ the **right child** of p . In $B = (N, L \cup R)$ a node $p \in N$ may have many direct predecessors q . The edges $(q, p) \in E$ are then called **references** from q to p . As an example see figure 1.2.

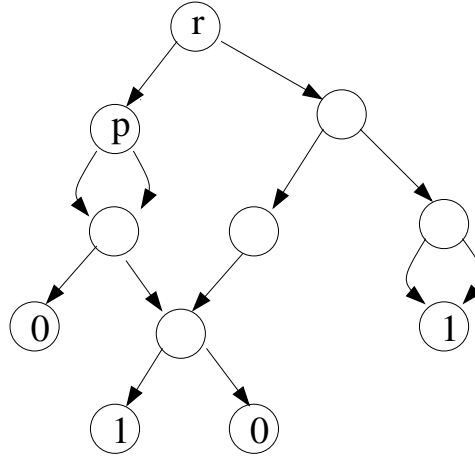


Figure 1.2: A binary tree B which allows multiple references. For instance it is $\text{LEFT}(r) = p$, $\text{LEFT}(p) = \text{RIGHT}(p)$, $\text{seq}(p) = 010010$, and $\text{seq}(B) = \text{seq}(r) = 0100101011$.

With each node $p \in N$ we can identify a string $\text{seq}(p) \in X^*$: If p is a leaf, then according to definition 2 we have $p \in X$ and can set $\text{seq}(p) = x$ (the string containing only x as its single letter). Since $(N, L \cup R)$ is acyclic, we can define for all inner nodes $p \in N$ recursively $\text{seq}(p) = \text{seq}(\text{LEFT}(p))\text{seq}(\text{RIGHT}(p))$ by setting $\text{seq}(\perp) = \varepsilon$. Finally, let r be the root of B . Then we define $\text{seq}(B) := \text{seq}(r)$. As an example, see again figure 1.2.

In our implementation, we maintain with each node $p \in N$ the length of $\text{seq}(\text{LEFT}(p))$. This allows to retrieve the i th symbol of the string $\text{seq}(B)$ by traversing the path from the root down to the corresponding leaf. We also maintain the value $\text{HEIGHT}(p)$ for each $p \in N$, which is defined to be the maximum length of a path from p to a leaf in $q \in N$. According to this definition, all leaves have height 0 and if we set $\text{HEIGHT}(\perp) = -1$ we further have

$$\text{HEIGHT}(p) = \max(\text{HEIGHT}(\text{LEFT}(p)), \text{HEIGHT}(\text{RIGHT}(p))) + 1$$

for all $p \in N$.

In order to keep the structure balanced, we use the AVL-condition for binary trees. Of course similar restrictions from other balanced trees will mostly be just as good:

Definition 3 A binary tree $B = (N, L, R)$ is an AVL-tree, if it fulfills the AVL condition, that is, for each $p \in N$ we have $|\text{HEIGHT}(\text{LEFT}(p)) - \text{HEIGHT}(\text{RIGHT}(p))| \leq 1$.

Algorithms for maintaining balance conditions in our trees during updates can be found in any introductory textbook on data structures and algorithms.

Algorithm 1.2 concatenation of persistent arrays represented as binary trees

function CONCATENATE(node: r, l) returns node

input:

two persistent arrays as AVL trees (with multiple references) given by their root nodes r and l respectively.

output:

returns the root node r' of a new AVL tree such that $\text{seq}(r') = \text{seq}(r) \cdot \text{seq}(l)$.

if $r = \perp$ **then return** l

end if

if $l = \perp$ **then return** r

end if

$\Delta := \text{HEIGHT}(r) - \text{HEIGHT}(l)$

if $\Delta > 1$ **then**

$l' := \text{CONCATENATE}(l, \text{LEFT}(r))$

if $\text{HEIGHT}(\text{LEFT}(r)) = \text{HEIGHT}(l')$ **then**

return $\text{NODE}(l', \text{RIGHT}(r))$

else

if $\text{HEIGHT}(l') - \text{HEIGHT}(\text{RIGHT}(r)) = 2$ **then**

if $\text{HEIGHT}(\text{RIGHT}(r)) = \text{HEIGHT}(\text{LEFT}(l'))$ **then**

return $\text{NODE}(\text{NODE}(\text{LEFT}(l'), \text{LEFT}(\text{RIGHT}(l'))), \text{NODE}(\text{RIGHT}(\text{RIGHT}(l')), \text{RIGHT}(r)))$

else

return $\text{NODE}(\text{LEFT}(l'), \text{NODE}(\text{RIGHT}(l'), \text{RIGHT}(r)))$

end if

else

return $\text{NODE}(l', \text{RIGHT}(r))$

end if

end if

else if $\Delta < -1$ **then**

 This case is essentially symmetric to the case $\Delta > 1$

 ...

else(we have $\Delta = 0$)

return $\text{NODE}(l, r)$

end if

end function

Every path in an AVL tree must fork at least every second level. So it can be shown that for all nodes p , we have $\text{HEIGHT}(p) \leq O(\log(\text{LENGTH}(p)))$.⁴

For the sake of completeness, we show the operations for concatenation \cdot and splitting (HEAD and TAIL), which are particularly easy to implement, run in time $O(\text{HEIGHT}(r))$, and allow us to implement the full interface of the ADT persistent array. For concatenation, see algorithm 1.2.

Here, $\text{NODE}(l, r)$ is a constructor which yields a newly created node p with $\text{LEFT}(p) = l$ and $\text{RIGHT}(p) = r$. The constructor also determines p 's correct values for $\text{HEIGHT}(p)$ and $\text{LENGTH}(p)$. Note that compared to algorithm 1.2, industrial-strength code is far more complex since it has to deal with many details such as memory management, representation of arbitrary large numbers for the LENGTH-attributes, and compactification of nodes for time and space efficiency.

We do not show that algorithm 1.2 indeed produces the desired output and that it maintains the AVL condition. Details can be found in [Knu 73]. Note that its running time is $O(|\text{HEIGHT}(r) - \text{HEIGHT}(l)|)$. This is important for the running time of splitting. Next we show the algorithm for HEAD, the algorithm for TAIL is symmetric.

Algorithm 1.3 Splitting of a persistent array represented as a binary tree

function HEAD(node: r , integer: i) returns node

input:

a persistent array as binary trees (with multiple references) given by its root node r , and an index i .

output:

returns the root node r' of a new binary tree such that $\text{seq}(r') = \text{HEAD}(\text{seq}(r))$.

if $r = \perp \quad \vee \quad i \leq 0$ **then return** \perp

else if $\text{HEIGHT}(r) = 0$ **then return** r

else if $\text{LEFT}(r) = \perp$ **then**

return HEAD(RIGHT(r), i)

else

if $i \leq \text{LENGTH}(\text{LEFT}(r))$ **then**

return HEAD(LEFT(r), i)

else

return CONCATENATE(LEFT(r), HEAD(RIGHT(r), $i - \text{LENGTH}(\text{LEFT}(r))$))

end if

end if

end function

⁴Better estimates can be obtained by observing that in the worst case, the lengths of the underlying sequences obey the recursion law of Fibonacci numbers.

Given a root node r of height h and an integer i , algorithm 1.3 will concatenate at most $h + 1$ binary trees of ascending height $\leq h$. Since the cost for a single concatenation is proportional to the difference of heights, the sum of the costs for all the subsequent concatenations telescopes and hence is bounded by $O(h)$.

We already discussed how the REVERSE-operation can be implemented in time $O(1)$ by maintaining two versions of each array. Since all the other operations from the interface (indexed element retrieval, concatenation, splitting and derived operations) run in time $O(h)$, where h is the maximum height of the roots of the involved AVL trees, and since each operation increases the maximum height at most by 1, we get the following result:

Proposition 1 *With a random access machine (RAM) with bounded word size for arithmetic, the ADT for persistent arrays can be implemented in such a way that each of its operations runs in time $O(\log n)$ if the arrays' lengths are representable within the word boundaries. Otherwise the time bound is $O(\min((\log n)^2, k^2))$. Here, n is the maximum length of the involved arrays, and k is the number of array operations that have been used in order to create the operand arrays. In particular, it is possible to simulate a program with k array operations in time $O(k^3)$ on a RAM.*

PROOF: Observe that the height of a node p in an AVL tree is at most $O(\log |\text{seq}(p)|)$. This yields the two logarithmic bounds. For long arrays with lengths not representable in a machine word, the general unit-cost assumption for integer arithmetic does not hold. For each node p along the path, the attribute $\text{LENGTH}(p)$ has to be maintained which might have bit-size proportional to $\text{HEIGHT}(p)$. We then get the bound $O((\log n)^2)$. The bound $O(k^2)$ follows from the fact that k operations can create AVL trees at least of height k and, again, the time for arithmetic.

The last part of the proposition follows by adding the costs $O(i^2)$ for k subsequent operations:

$$\sum_{i=1}^k c \cdot k^2 = O(k^3)$$

□

The time bounds can be understood in two different ways: On one hand, we might want to replace the built-in array type of a programming language by the introduced persistent array, or just use persistent arrays as an ADT. If we use the arrays in the normal way, in the sense that the array lengths grow only polynomially in the size of the input, the first time bound bounds the additional costs for each array operation by logarithm of the maximum length of the operands. Then the array operations contribute only a poly-logarithmic factor to the program's running time.

On the other hand, there might be situations when the array length can grow enormously, even exponentially during course of computation. The second part of proposition 1 states that we do not need to care much about the actual array lengths as long as we restrict ourselves to the operations of the ADT persistent array. For instance, if we count each array operation which occurs during the course of

computation only once, the actual running time of our implementation will still only have an overhead which depends polynomially — that is $O(k^3)$ — on the number k of array operations.

For some applications, it might be worthwhile to export some implementation-specific details such as the LEFT and WRITE operations on inner nodes. This sometimes saves a logarithmic factor for certain binary search schemes. We extend the ADT PERSISTENT ARRAY by the following operations:

1. LEFTPART(A): returns (non-deterministically) an array representing a prefix A 's sequence.
2. RIGHTPART(A): returns (non-deterministically) an arrays representing a suffix A 's sequence.
3. SOMELEMENT(A): returns (non-deterministically) an element of the array A .

All three operations shall run in $O(1)$ time. The operation SOMELEMENT can easily be implemented by storing additional information in the inner nodes of the binary tree.

Note that for two arrays A and B , both representing the same sequence, the result of the operations above is not necessarily the same. However, it always holds

$$A = \text{LEFTPART}(A) \cdot \text{RIGHTPART}(A) .$$

Also we require for each array A of length n , that after $O(\log n)$ subsequent LEFTPART or RIGHTPART operations (in any possible combination) the resulting array has length 0 or 1.

1.4.3 Homomorphisms on the Free Monoid

Assume we have an array of numbers, and we wish to maintain the sum over all its elements during array operations such as insertion, concatenation, splitting, etc. It would be very costly if, after each operation, we had to iterate over all elements in order to recompute their sum. For instance, if $x \in X^*$ with $X = \mathbb{N}$ is a finite sequence $x = x_0 \dots x_{n-1}$ of numbers $x_i \in \mathbb{N}$, we would like to query **segment sums**: that is, given two indices $i, j \in \{0..n-1\}$, we ask for the value

$$\text{SEGMENTSUM}(x, i, j) := h \left(\sum_{k=i}^j x_k \right) .$$

If h is a function which maps any sequence $y \in \mathbb{N}^*$ to the sum of *all* elements of y , then:

$$\text{SEGMENTSUM}(x, i, j) = \text{TAIL}(\text{HEAD}(x, j+1), i)$$

Hence it would be nice if the images $h(x)$ of a sequence x could be maintained under the operations of the ADT PERSISTENT ARRAY . As it turns out, we can do this efficiently whenever h maps into a set M which is equipped with an associative operation \cdot_M and a neutral element e_M , where in our example $M = \mathbb{N}$, $\cdot_M = +$, and $e_M = 0$. Such a structure $M = (M, \cdot_M, e)$ is called a **monoid**. The second requirement is that h is a **homomorphism** on the free monoid X^* :

$$h(x \cdot y) = h(x) \cdot_M h(y), \quad \text{for all } x, y \in X^*$$

Let X be a (possibly infinite) set of symbols, $M = (M, \cdot, e)$ be a monoid and $h : X^* \rightarrow M$ be a homomorphism. Since X^* is the free monoid which is generated by X , h is uniquely defined by all the values $h(x)$ for $x \in X$. For strings $v \in X^*$, many properties can be defined in terms of homomorphic images. Besides of sums, another simple example are maxima: If $X = \mathbb{R}$, the maximum of a sequence $x_0 \cdots x_{n-1}$, $n \geq 0$ of real numbers is the image of $x_0 \cdots x_{n-1}$ under the homomorphism $h : \mathbb{R}^* \rightarrow (\mathbb{R} \cup -\infty, \max, -\infty)$ which is defined by $h(x) = x$ for all $x \in \mathbb{R}$. In chapter 3 we will see other array parameters and transformations that can be expressed this way.

For computing the homomorphic image $h(v)$ of a string $v \in X^*$, it is sufficient to know only the restriction of h on X , provided that we also know the multiplication and the neutral element of M . Thus when we say — in the context of a computer program — that a homomorphism $h : X^* \rightarrow M$ is given, we actually mean that we are given a triple (h', \cdot, e) . Here the operation \cdot is the multiplication of the monoid M , e is its neutral element, and h' is the restriction of $h : X^* \rightarrow M$ to the set X of single elements.

As in our segment-sum example, sometimes a homomorphism $h : X^* \rightarrow M$ is given in advance, and we want to maintain the homomorphic image $h(v)$ for all strings $v \in X^*$ which occur during the course of computation. In a binary tree representation B , we can achieve this by labelling each node p in B with $h(\text{seq}(p))$. Therefore, we modify the constructor of inner nodes p in such a way that, besides of $\text{LEFT}(p)$ and $\text{RIGHT}(p)$, the constructor also gets the multiplication $\cdot : M \times M \rightarrow M$ as a parameter. Then, by homomorphism:

$$h(\text{seq}(p)) = h(\text{seq}(\text{LEFT}(p))) \cdot h(\text{seq}(\text{RIGHT}(p)))$$

It follows:

Proposition 2 *Let $h : X^* \rightarrow M$ be a fixed monoid homomorphism. By using AVL trees (with multiple references), it is possible to maintain homomorphic images $h(A)$ of all persistent arrays A in such a way that the additional costs stem solely from computing $h(x)$ for newly created array elements $x \in X$ and from multiplications in M for the inner nodes. The latter is bounded by $O(\min(\log n, k))$ for each single operation, where n is the maximum length of the involved arrays, and k is the number of array operations that have been used in order to create operands. In particular, it is possible to simulate k steps of a random access machine — equipped with array operations on the ADT PERSISTENT ARRAY — in time $O(k^3)$ plus the time needed for $O(k^2)$ multiplications in M .*

The proposition is similar to proposition 1 and additionally states that maintaining homomorphic images for a fixed homomorphism h does not do any harm as long as the costs for mapping of single elements and for multiplication in the range of h can be neglected. Note that this proposition extends easily to any constant number of homomorphisms.

Sometimes we do not want to maintain images under previously fixed homomorphisms. What we want is to compute an array's A image $h(A)$ for a given homomorphism h . This can be accomplished by traversing the binary tree which represents A :

Proposition 3 *Let $h : X^* \rightarrow M$ be a fixed monoid homomorphism. For a particular persistent array A which occurs in the k th step of a computation, it is possible to compute its homomorphic image in such a way that the costs stem solely from multiplications in M and from computing $h(x)$ for the leaf elements of the AVL tree (with multiple references) which represents A . The number of multiplications is bounded by the number of inner nodes of the tree, which is $O(\min(k^2, |A|))$.*

Note that while homomorphisms are useful, their usage is — compared to the standard operations on persistent arrays — somewhat restricted: Either we maintain images for a fixed finite set of homomorphism during the course of computation, or we halt the computation and compute homomorphic images afterwards. The propositions above give reasonable bounds on the number of operations. Note, however, that it is inherently more difficult to derive appropriate time bounds if the computation of homomorphism is interwoven with the usual array operations from the ADT. For instance, for a given n

```

A := [1]
let  $h : \{0, 1\}^* \rightarrow \{0, 1\}^*$  be the homomorphism defined by  $h(0) = 0, h(1) = 11$ 
for  $i = 0..n - 1$  do
     $A := h(A)$ 
end for

```

runs in time polynomial in n only when we assume appropriate copy semantics in the computation of $h(x)$. If each time the array which represents $h(x)$ is newly created, then the running time can actually be exponential in n .

On the other hand, the same resulting string will be computed by

```

A := [1]
for  $i = 0..n - 1$  do
     $A := A \cdot A$ 
end for

```

always in time polynomial in n .

1.4.4 Impossibility of Pairing

Given two strings $v = x_1 \cdots x_n$, $w = y_1 \cdots y_n$, both of length n , we would like to construct a persistent array C for the string $v \times w := (x_1, y_1) \cdots (x_n, y_n) \in (X \times X)^*$. The operation which maps v, w to $v \times w$ is called **pairing**. A representation of the string $v \times w$ in form of a persistent array C would permit to build many other operations on top of it: For instance, if v, w are two n -vectors of natural numbers, we could use a homomorphism which maps $v \times w$ to the n -vector of element-wise sums. This would allow vector addition even for vectors of exponential length — as long as the persistent array representation is of polynomial size. Another example is to use C in order to decide if v, w are equal: All we have to do

is to apply the homomorphism $h : (X \times X)^* \rightarrow \mathbb{B}$ where $\mathbb{B} = (\{0, 1\}, \wedge, 1)$ is the boolean monoid with \wedge (logical AND) as multiplication and 1 as neutral element, and h is defined by

$$h((x, y)) := \begin{cases} 1, & \text{iff } x = y \\ 0, & \text{else} \end{cases}$$

for $(x, y) \in X \times X$.

The operation which maps v, w to $v \times w$ is called **pairing**. Sadly, even though the operation is useful, it is not possible to perform pairing in polynomial time if we use binary trees for persistent arrays and measure the input size by its number of nodes. The reason is that pairing might cause the number of nodes to grow exponentially, as we will see below.

Proposition 4 *Choose $X = \{0, 1, *\}$, $v = (0^k 1)^k$ and $w = *(0^{k+1} *)^{k-1} 0$. Then any binary array structure which represents $v \times w$ has at least $k/2$ inner nodes.*

PROOF: Both, v and w have length $k^2 + k$, thus $v \times w$ is well-defined. For instance, for $k = 3$, $v \times w$ is the string

$$(0, *) (0, 0) (0, 0) (1, 0) (0, 0) (0, *) (0, 0) (1, 0) (0, 0) (0, 0) (0, *) (1, 0) .$$

By substituting $(0, 0) \mapsto 0, (1, 0) \mapsto 1, (0, *) \mapsto *$ we get a string x , which is for $k = 3$:

$$*0010*0100*1$$

For $k = 6$, x would be

$$*0000010*0000100*0001000*0010000*0100000*1 .$$

The string x contains all possible strings $0^i * 0^{k-i-1} 1$, $0 \leq i < k$, as infixes, each one exactly once. Assume now that $v \times w$ is represented by a binary tree (allowing multiple references). Then x can be represented by essentially the same tree, which we call B . If p is a node of B which represents a string containing at least one of the $0^i * 0^{k-i-1} 1$, then p cannot be referred more than once in B : if so, then $0^i * 0^{k-i-1} 1$ would be contained more than once in $x = \text{seq}(B)$ which contradicts that the position of any $0^i * 0^{k-i-1} 1$ in x is unique. It follows that there is at most one reference to any node p with $|\text{seq}(p)| \geq 2k + 1$. It follows that each path originating at B 's root starts with nodes p which are only referenced once in B , at least until $\text{seq}(p) \leq 2k$. So the top of B is essentially a (true) binary tree with at least $k^2/2k = k/2$ many nodes. Since B was chosen to be essentially the same as the arbitrarily chosen binary tree for $v \times w$ (both allowing multiple references), this proves our claim. \square

Corollary 5 *It is not possible to implement pairing in polynomial time for arrays as binary trees (with multiple references).*

PROOF: Set $k = 2^n$. By fast exponentiation, persistent arrays (as binary trees with multiple references) for the strings v, w from proposition 4 can be constructed in time polynomial in n and thus have only a polynomial number of nodes. By proposition 4 any representation of $v \times w$ as a binary tree has at least $\Omega(2^n)$ many nodes and thus cannot be constructed from the binary trees (with multiple references) for v and w in polynomial time. \square

But would other implementations of the ADT PERSISTENT ARRAY be of any benefit? That is, can we still simulate a machine equipped with array concatenation and pairing in polynomial time in such a way that afterwards we can compute homomorphic images efficiently? The answer is negative in the following sense:

Corollary 6 *For $k = 2^n$, the arrays v, w in the proof of proposition 4 can be constructed by a program equipped with array concatenation with a polynomial (in n) number of steps. However:*

1. *It is impossible to represent $v \times w$ in such a way that there is a generic algorithm which, for any monoid homomorphism $h : (X \times X)^* \rightarrow M$, computes $h(v \times w)$ by a polynomial number of multiplications, starting from the images $h(X) \subseteq M$.*
2. *Let $h : (X \times X)^* \rightarrow M$ be a homomorphism such that any element of M generated by $h(X) \subseteq M$, and a program which uses a polynomial number of multiplications (in M), can be also be computed in polynomial time. Then h can be chosen in such a way that $h(v \times w)$ cannot be computed in polynomial time.⁵*

PROOF: The proof is clear by choosing h to be the identity map on $(X \times X)^*$, and representing the images of h as binary trees with multiple references. (More formally, h is a homomorphism between two different representations of $(X \times X)^*$, where structures representing the same string are identified). Firstly, any generic algorithm which computes $h(v \times w)$ as shown in the first part of the corollary must fail by proposition 4. Secondly, observe that h fulfills the requirements of the second part of the corollary by proposition 1. Hence polynomial-time construction of $h(v \times w)$ contradicts proposition 4. \square

Thus pairing cannot be a polynomial time operation for any polynomial-time implementation of the ADT PERSISTENT ARRAY which allows efficient computation of homomorphic images. The results suggest that it is better to exclude pairing from the interface of persistent arrays. It also indicates that deciding equality (in terms of the sequence of elements) of persistent arrays represented as binary trees A, B is difficult when multiple references are involved (thus that the lengths can grow exponentially). However, in section 3.1.1 we will give a partial solution by describing a probabilistic equality test.

By using the same construction as in proposition 4 we get:

⁵Note that the time for computing $(x, y) \mapsto h((x, y))$ for single pairs $(x, y) \in X \times X$ can be ignored since the number of elements of $X = \{0, 1, *\}$ is finite.

Corollary 7 *For each $n \geq 0$, there exist strings $v, w \in \{0, 1\}^*$ whose representation as binary trees with multiple references is constructable in time polynomial in n , such that each binary tree representing the sum of v and w (interpreted as binary numbers) needs an exponential number of nodes.*

1.5 Related Work

Dynamic arrays with concatenation and splitting and their implementation as balanced trees are well-known. They can be found in many textbooks on algorithms and data structures, see [Knu 73] for an implementation as AVL trees, or [CLRS 01] for an implementation as red-black trees. In [Knu 73], Knuth credits C.A. Crane’s 1972 thesis for the first description of concatenating and splitting AVL trees. In the literature for functional data structures and also for graph algorithms, fully persistent arrays belong to the fundamental data structures for a long time. The possibility of maintaining homomorphic images is a fact which is rarely explicitly mentioned. Still we believe that our explicit treatment is worthwhile, since the homomorphism technique can be considered as a programming interface which generalizes from many data structures. As an example see the J. Burghardt’s paper [Bur 02] about segment sum queries. String homomorphisms have also been studied as a tool for parallelizing algorithms. Our explicit upper bounds for polynomial time simulation of persistent arrays in connection with proposition 4 about the impossibility of pairing are new in this thesis.

Chapter 2

Graph Algorithms and Persistent Arrays

In the first chapter we described the basic properties of persistent arrays. In particular we stressed that their representation allows sharing of subarrays. Furthermore, we stated that persistence makes the design of algorithms easier. The purpose of this chapter is to justify this claim in the context of graph algorithms.

The main idea behind the data structures and algorithms in this chapter is to represent paths as persistent arrays. Persistent arrays provide an interface for obtaining other information on a given path by means of homomorphic images, which might be useful as a general interface in programming libraries. We have already seen in section 1.4.3 how homomorphic images can be maintained efficiently for persistent arrays. For example, assume that a train route is represented as a string $v = s_0 d_1 s_1 \dots s_{n-1} d_n s_n$, where each s_i is a station and d_i is the distance from s_{i-1} to s_i . Then the following information can be obtained in terms of homomorphisms:

1. The total distance $\sum_{i=1}^n d_i$.
2. The maximum distance $\max(d_0, \dots, d_n)$ in between two stations.
3. A string $s_{i(0)} s_{i(1)} \dots s_{i(k)}$, $0 \leq i(0) < i(1) < \dots < i(k) \leq n$ containing only particular stations, for instance stations where the train halts or stations with special facilities for maintenance etc.

In this chapter, we will see how paths, represented as persistent arrays, can be maintained within several known graph algorithms and data structures:

The first section deals with dynamic tree algorithms. Dynamic data structures for (undirected) trees are fundamental for many dynamic graph algorithms, for instance for solving dynamic connectivity problems. We will use Henzinger and King's ET trees as a data structure for dynamic trees (see [HLT 01]). ET trees are easy to implement but do not maintain path information or global tree parameters such as center. We will show how persistent arrays can be used in order to maintain path information and center under update operations while retaining the simplicity of the data structure. Our resulting data structure has not the asymptotically best performance, but is easy to implement and updates and queries still run

in poly-logarithmic time.

In the concluding sections of this chapter we will prepare the semiring framework for using persistent arrays for all-pairs-shortest-path (APSP) algorithms. We also show how the successor matrix representation of a solution of the APSP problem can be efficiently transformed into a matrix of persistent arrays.

2.1 Paths in Dynamic Trees and Forests

First, we settle some notation: Let $G = (N, E)$ be an undirected graph. G is a **forest** if G is acyclic (but not necessarily connected). If G is a forest and connected, then we call G a **tree**.

Throughout the entire section, a path π is a finite sequence $\pi = a_0 \dots a_k$ of nodes $a_i \in N$, such that for $0 \leq i < k$ the nodes a_i, a_{i+1} are adjacent to each other. We write $\text{src}(\pi)$ for a_0 and $\text{dst}(\pi)$ for a_k . The length of a path π is its number k of traversed edges. Our algorithms represent paths as persistent arrays, thus making it possible to join two given paths π, π' in logarithmic time (depending on their length). Also, we can retrieve the i th node of a path in logarithmic time. Note that joining π and π' only yields a path if $\text{dst}(\pi) = \text{src}(\pi')$, and that we actually remove either $\text{dst}(\pi)$ or $\text{src}(\pi')$ before concatenation. The notation $\pi : p \rightarrow_G q$ means that π is a path (in the underlying graph G) with $\text{src}(\pi) = p$, $\text{dst}(\pi) = q$. The subscript G will be usually omitted.

From now, let $T = (N, E)$ be a tree. For each pair of nodes $p, q \in N$ there exists a unique simple path $\pi : p \rightarrow q$, denoted by $\pi(p, q) := \pi$. The main idea of this section is to compose simple paths in order to gain another simple path:

Proposition 8 *Let $p, q, r \in N$ and let $a_0 \dots a_k = \pi(r, p)$, $b_0 \dots b_m = \pi(r, q)$. Then there exists no pair of indices $i < j \in \{0 \dots \min(k, m)\}$ such that $a_i \neq b_i$ and $a_j = b_j$.*

PROOF: Assume such a pair $i < j$ exists. Since $a_0 = b_0 = r$ there exists a maximum $s < i$ with $\pi_s = \pi'_s$. Because of $\pi_0 = \pi'_0 = r$ there exists a minimum $t > i$ with $\pi_t = \pi'_t$. Thus, the path a_s, a_{s+1}, \dots, a_t connected with b_t, b_{t-1}, \dots, b_s forms a cycle, which contradicts that T is a tree. \square

From proposition 8 we see that for $a_0 \dots a_k = \pi(r, p)$ and $b_0 \dots b_m = \pi(r, q)$, the maximum index s with $a_s = b_s$ is well-defined. It divides the paths $\pi(r, p), \pi(r, q)$ into a common subpath and two node-disjoint subpaths which then form the (unique) simple path $\pi(p, q)$:

Corollary 9 *Let be $p, q \in N$ and $a_0 \dots a_k = \pi(r, p)$, $a_i \in N$, $0 \leq i \leq k$ and $b_0 \dots b_m = \pi(r, q)$, $b_i \in N$, $0 \leq i \leq m$. Choose $s = 0 \dots \min(k, m)$ such that $a_0 \dots a_s = b_0 \dots b_s$ is the longest common prefix of both paths. Then $\pi(p, q) = a_k a_{k-1} \dots a_s b_{s+1} b_{s+2} \dots b_m$.*

The proof is immediate by observing that $a_k a_{k-1} \dots a_s$ and $b_s b_{s+1} \dots b_m$ have only the node $a_s = b_s$ in common. This node $a_s = b_s$ from above is denoted by $\text{MEET}(p, q, r)$, it is the intersection of the three paths connecting p, q, r .

Now, for two simple paths $\pi = \pi(r, p)$, $\pi' := \pi(r, q)$ with k and m nodes respectively, we can compute $\text{MEET}(p, q, r)$ in $O((\log \min(k, m))^2) + O(\log \max(k, m))$ time. Within the same time bounds, we can compose $\pi(r, p)$ and $\pi(r, q)$ in such a way that we gain the (unique) simple path $\pi(p, q)$. The resulting algorithm is algorithm 2.1 for composing two simple paths to another simple path, which implicitly computes $\text{MEET}(p, q, r)$.

Algorithm 2.1 Path Composition

function COMPOSESIMPLEPATHS(π, π')

Input: two simple paths $\pi : r \rightarrow p$, $\pi' : r \rightarrow q$, represented as persistent arrays

Output: the simple path $\pi(p, q)$

$k := \min(\text{LENGTH}(\pi), \text{LENGTH}(\pi'))$

$i := 0; j := k - 1$

$\pi := \text{HEAD}(\pi, k)$

$\pi' := \text{HEAD}(\pi', k)$

while $i \neq j$ **do**

$s := \lceil (j - i) / 2 \rceil$

if $\pi[s] = \pi'[s]$ **then**

$i := s$

else

$j := s - 1$

end if

end while

$(\pi[i] = \pi'[i] \text{ is now } \text{MEET}(p, q, r))$

return REVERSE(TAIL(π, i)) · TAIL($\pi', i + 1$)

end function

In order to achieve the time bounds, observe that the binary search in the first part of algorithm 2.1 accesses the arrays $O(\log \min(k, m))$ times, while each access needs $O(\log \min(k, m))$ time. Hence the first term $O((\log \min(k, m))^2)$. Finally, splitting and concatenation need $O(\log \max(k, m))$ time each.

2.1.1 Path Queries on Static Trees

Before we turn to the dynamic case, we will consider path queries on a given static tree $T = (N, E)$ with n nodes. The goal of a path query $\text{PATH}(p, q)$ is, given two nodes p, q , to yield the simple path $\pi(p, q)$. A direct consequence of the path-composition algorithm 2.1 is:

Proposition 10 *Given a fixed $r \in N$ and a table of all $\pi(r, p)$, $p \in N$, represented as persistent arrays. Then for any pair of nodes $p, q \in N$, a persistent array representing $\pi(p, q)$ can be constructed in time $O((\log m)^2)$, where m is the length of the longest simple path in T .*

For most “natural” graph representation of a tree $T = (N, E)$ and any given $r \in N$, such a table $\pi(r, p)$, $p \in N$ can be constructed efficiently by using DFS traversal. For instance, a natural representation of a tree T is by a collection of adjacency lists, which for each node store the list of adjacent nodes. This representation needs — since T is a tree — $\Theta(n)$ space. Adjacency lists do not support path queries, but they permit traversal of the nodes N depth-first (DFS-order) in $O(n)$ time, starting from $r \in N$.

Algorithm 2.2 Construct Table of Paths rooted in r

function CONSTRUCTTABLE(graph G , node r)

input:

An undirected tree $G = (N, V)$ and a arbitrary root node $r \in N$. It is assumed that G is represented in a natural way which supports DFS-traversal without additional overhead

result:

constructs the table $\text{path}[]$ with $\text{path}[p] = \pi(r, p)$

⟨ mark all nodes $p \in N$ as unvisited ⟩

VISIT($r, \epsilon, 0$)

end function

function VISIT(node p , Array P) returns number of visited nodes

if ⟨ p is not already visited ⟩ **then**

⟨ mark p as visited ⟩

$P := \text{CONCATENATE}(P, [p])$

$\text{path}[p] := P$

for ⟨ all nodes q adjacent to p ⟩ **do**

VISIT(q, P)

end for

end if

end function

By using DFS traversal we can now transform T 's representation (as adjacency lists) into the table $\pi(r, p)$ by bookkeeping the simple path from r to the currently visited node p . By proposition 10, the resulting table $\pi(r, p)$ permits path queries in $O((\log n)^2)$ time. Algorithm 2.2 will construct $\pi(r, p)$ for all $p \in N$ simultaneously in time $O(n \log n)$, where the additional logarithmic factor stems from the time for concatenating persistent arrays. Note that the space requirements for representing the table $\pi(r, p)$ in memory is naturally bounded by the running time of algorithm 2.2, which is $O(n \log n)$.

As we see, a representation of a tree T (which allows DFS traversal without additional overhead except the time for visiting the nodes), can be transformed into a representation such that:

1. The new representation needs $O(n \log n)$ space.
2. Path queries and MEET-queries need time $O((\log m)^2)$, where $m \leq n$ is the length of the longest simple path in T .
3. The transformation needs $O(n \log n)$ time.

2.1.2 Path Queries on Dynamic Trees

A dynamic forest is a data structure which represents a collection of dynamic trees, allowing updates on the set of edges. The two most important update operations are linking two formerly disconnected trees by a newly introduced edge, and splitting a tree by removing one of its edges.

An interesting application for trees and forests as dynamic data structures are computer networks – their topologies are often trees themselves. Other applications of dynamic forests arise in dynamic graph algorithms, such as maintaining spanning trees for connected components or for solving several connectivity problems (see [HLT 01]).

Many data structures have been proposed for dynamic trees, among them Heinzinger and King's ET tours, which are particularly simple to implement. More elaborate data structures rely on a recursive decomposition of trees and are far more complicated to implement than ET trees (for instance topology tree [Fre 85] and top trees [AHLT 03]). While topology trees and top trees are very efficient and versatile, nevertheless ET trees suffice for many purposes. In their standard version, all three data structures allow updates in $O(\log n)$ time, but only top trees and topology trees maintain path information and global graph parameters such as diameter. In the remainder of this section we want to fill the gap between the highly efficient and versatile data structures for dynamic trees and the much easier to implement ET trees: We will show how ET trees can be modified in such a ways that they allow to maintain paths and diameters. Additional path parameters like maximum edge-value or total distance can then be maintained in terms of homomorphic images (see section 1.4.3). The resulting data structure retains the simplicity of ET trees, and still implements trees with updates and queries in poly-logarithmic time.

The running time per update and path query will be $O(\log n (\log m)^2)$, where n is the size of the tree(s), measured in the number of nodes, and m is the maximum number of nodes for a simple path in the forest. For updates, this is by a factor $(\log m)^2$ worse than the best known data structures; for path queries (when paths are represented as persistent trees) it is worse by a factor of $\log m$. Our interest in using ET trees is legitimated since they permit simple implementations for prototyping or for building a test bench for the more elaborated data structures.

Let $F = (N, E)$ be a forest consisting of the maximal trees (in the meaning of not being proper subtrees included within other trees in F) T_1, \dots, T_k . Deleting any edge from F will increase the number

of maximal trees by one. For disconnected $p, q \in N$, adding the edge (p, q) will yield another forest $F' = (N, E \cup \{(p, q)\})$ with the number of maximal trees reduced by one. We have the following fundamental update operations:

1. $\text{LINK}(p, q)$: Given two disconnected nodes $p, q \in N$, returns the forest $F' = (N, E \cup \{(p, q)\})$.
2. $\text{UNLINK}(p, q)$: Given two adjacent nodes $p, q \in N$, returns the forest $F' = (N, E \setminus \{(p, q)\})$.

Next to update operations, we want to support the following queries:

1. $\text{ISCONNECTED}(p, q)$: Given two nodes $p, q \in N$, determines whether they are connected.
2. $\text{PATH}(p, q)$: Given two connected nodes $p, q \in N$, returns the simple path $\pi(p, q)$.
3. $\text{ISBETWEEN}(p, q, r)$: Given three connected nodes $p, q, r \in N$, determines whether r lies on the simple path $\pi(p, q)$.
4. $\text{MEET}(p, q, r)$: Given three connected nodes $p, q, r \in N$, returns the (unique) intersection point of the three paths between each pair of the nodes p, q, r .
5. $\text{CENTER}(p)$: Given a node $p \in N$, returns a center of T_i , where T_i is the maximal tree which contains p .
6. $\text{DIAMPATH}(p)$: Given a node $p \in N$, returns a simple path of maximum length in T_i , where T_i is the maximal tree which contains p .
7. $\text{DIAM}(p)$: Given a node $p \in N$, returns a diameter path of T_i , where T_i is the maximal tree which contains p .

For the definition of diameter, diameter path, and center see section 2.1.3.

MEET and ISBETWEEN can be implemented in terms of path queries: For computing MEET we can use path composition, while for $\text{ISBETWEEN}(p, q, r)$, we find that r lies on $\pi(p, q)$ if and only if $\text{MEET}(p, q, r) = r$.

In order to allow update operations and the remaining queries, we linearize a tree by transforming it to one of its Euler tours, which in turn is represented as a so-called ET tree: In a directed graph, an **Euler tour** is a closed path which traverses every edge exactly once. An Euler tour of an undirected tree T is then an Euler tour of its directed equivalent $T' = (N, \{(p, q) : \{p, q\} \in T\})$. There always exists at least one Euler tour, since in T' all nodes have the same in-degree and out-degree. For a pictorial representation of a tree's Euler tour see figure 2.1.

Proposition 11 *Let $T = (N, E)$ be a non-empty tree. The set of Euler tours for a tree T can be defined recursively as the smallest set fulfilling the following properties:*

- (i) If $T = (\{p\}, \emptyset)$, then the path $\pi = pp$ is the only Euler tour in T .

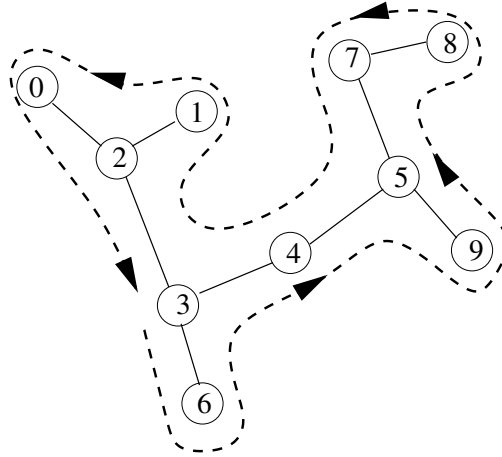


Figure 2.1: An Euler tour around an undirected tree

- (ii) Let be $T = (N_1 \cup N_2, E_1 \cup E_2 \cup \{p, q\})$ such that $T_i = (N_i, E_i)$, $i = 1, 2$ are trees and $N_1 \cap N_2 = \emptyset$, $p \in N_1$, $q \in N_2$. Let be $s_1 \in N_1$ and let $\pi_1 : s_1 \rightarrow p$, $\pi'_1 : p \rightarrow s_1$ be paths in T_1 such that $\pi_1 \pi'_1$ is an Euler tour in T_1 . Similarly, let be $s_2 \in N_2$ and let $\pi_2 : s_2 \rightarrow q$, $\pi'_2 : q \rightarrow s_2$ be paths in T_2 such that $\pi_2 \cdot \pi'_2$ is an Euler tour in T_2 . Then $\pi_1 \cdot (p, q) \cdot \pi'_2 \cdot \pi_2 \cdot (q, p) \cdot \pi'_1$ is an Euler tour of T .

The operation \cdot joins two paths π_0, \dots, π_m and π'_0, \dots, π'_k with $\pi_m = \pi'_0$ to $\pi_0, \dots, \pi_{m-1}, \pi'_0, \dots, \pi'_k$. Proposition 11 yields an equivalent definition of Euler tours in terms of a recursive decomposition of trees. We can represent each maximal tree $T_i = (N_i, E_i)$ of the forest F by the sequence of directed edges of some (arbitrarily chosen) Euler tour of T_i . If now we store the Euler tours in AVL trees, in a similar way as we implemented persistent arrays, we can split and concatenate Euler tours in logarithmic time. Since an Euler tour traverses each directed edge exactly once, the AVL tree has no multiple references to any of its inner nodes. This allows us to maintain **uplinks** from each inner node (except the root node) to its unique successor. The resulting structure is called an **ET tree**. In ET trees we can set the uplinks in the constructor calls of algorithms 1.2 and 1.3. Note that since this modification causes the constructor to operate destructively on its children, ET trees are not persistent.

A forest $F = (N, E)$ is now represented by a collection of ET trees ET_i , one for each of F 's maximal trees $T_i = (N_i, E_i)$. In addition to the ET trees, we will maintain a dictionary which, for each directed edge (p, q) with $\{p, q\} \in E$, allows us to find the corresponding leaf node in F 's collection of Euler trees. The dictionary is organized in such way that, given a node $p \in N$, it returns an arbitrary directed edge (p, q) with $\{p, q\} \in E$ or \perp if no such edge exists. Also, given a directed edge (p, q) with $\{p, q\} \in N_i$ the dictionary returns the leaf labeled with (p, q) of the corresponding ET tree ET_i . As an example, let us consider figure 2.2: We want to test whether or not the points $p = 5$ and $q = B$ are connected. Assume that for the nodes 5 and B the dictionary returns the edges $(5, 9)$ and (B, A) , respectively. The uplinks from edge $(5, 9)$ will allow us to walk upward in ET_1 via the nodes $e_{16}, e_9, e_4, e_2, e_1$. At the same time, the uplinks from edge (B, A) will allow us to walk upward in ET_2 via the nodes e_{22}, e_{21} . Since e_1 and

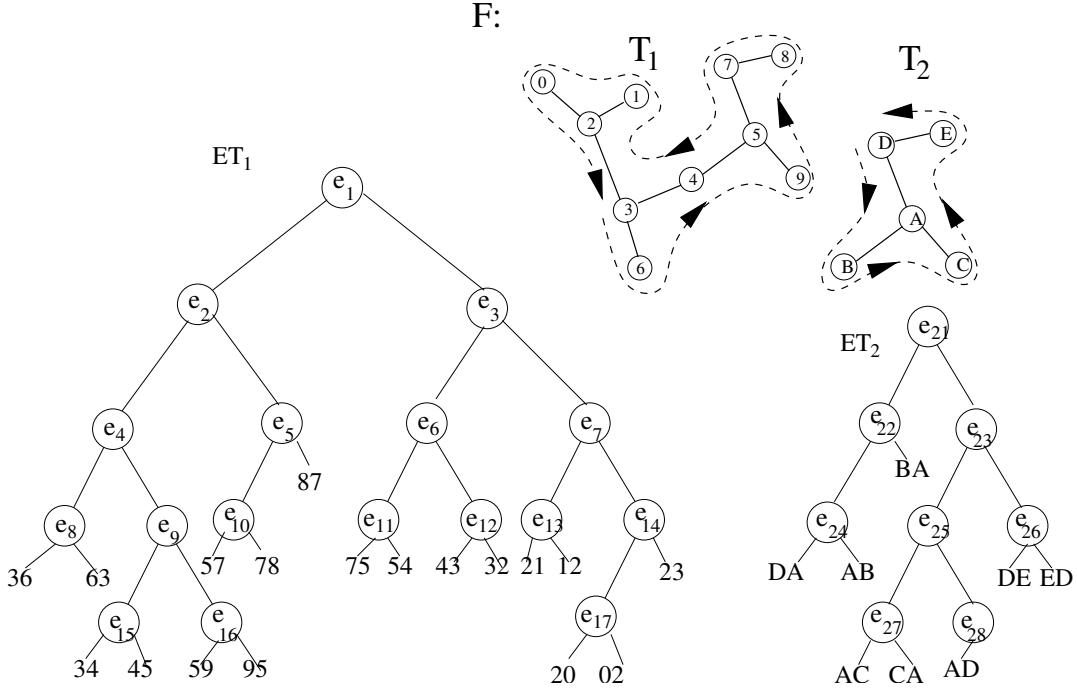


Figure 2.2: A forest F consisting of two maximal trees T_1, T_2 and two corresponding ET trees

e_{21} are the roots of two distinct ET trees, p and q are disconnected in F .

After having discovered that p and q are disconnected, we want to join both trees by introducing the edge $\{p, q\}$ to the forest F . Again, we find two corresponding paths to the root nodes of their respective ET trees, but this time we memorize the positions of the leaf nodes from where we started. This allows us to use the splitting operation **HEAD** and **TAIL** in order cut the Euler tour at appropriate positions for $p \in T_1$ and $q \in T_2$ and then to concatenate the results in order to obtain an ET tree for the tree $T = (N_1 \cup N_2, E_1 \cup E_2 \cup \{p, q\})$.

Indeed, proposition 11 shows us how to maintain Euler trees during updates through concatenating and splitting. Also, by using the uplinks in the ET trees, we can determine in logarithmic time whether any two nodes are connected.

For an inner node e of an ET tree ET_i , the edges which are contained in the leaves of the subtree at e form a (not necessarily simple) path in forest's corresponding tree T_i . We call this path the **Euler subtour** at e , denoted by $\text{st}(e)$. In order to allow path queries, we maintain for each e the simple path $\pi(\text{src}(\text{st}(e)), \text{dst}(\text{st}(e)))$, represented as a persistent array.

Let e be the inner node of an ET tree ET_i . If e is a leaf, then $\text{st}(e)$ consists solely of the directed edge (p, q) which e contains, and $\pi(\text{src}(\text{st}(e)), \text{dst}(\text{st}(e))) = \text{st}(e) = pq$. If e is an inner node of ET_i with only one child e' , then $\text{st}(e) = \text{st}(e')$ and we get $\pi(\text{src}(\text{st}(e)), \text{dst}(\text{st}(e))) = \pi(\text{src}(\text{st}(e')), \text{dst}(\text{st}(e')))$.

Finally, we have the case that e_1, e_2 are the left and the right children of e . Then $\text{src}(\text{st}(e)) = \text{src}(\text{st}(e_1))$, $\text{dst}(\text{st}(e)) = \text{dst}(\text{st}(e_1))$ and $\text{dst}(\text{st}(e_1)) = \text{src}(\text{st}(e_2))$. Hence we can compute $\pi(\text{src}(\text{st}(e)), \text{dst}(\text{st}(e)))$ by composing $\pi(\text{src}(\text{st}(e_1)), \text{dst}(\text{st}(e_1)))$ and $\pi(\text{src}(\text{st}(e_2)), \text{dst}(\text{st}(e_2)))$. Using algorithm 2.1, this needs $O(\log m)$ time per inner node e , where m is the maximum number of edges of any simple path. Hence, each update operation takes $O(\log n(\log m)^2) \leq O((\log n)^3)$ time.

Let $p, q \in N$ be connected. In order to extract the simple path $\pi(p, q)$, we first find the corresponding ET tree ET_i together with appropriate positions of p, q within the Euler tour. By splitting and concatenating ET_i appropriately, we obtain an ET tree ET' with root node r such that $\pi(\text{src}(\text{st}(r)), \text{dst}(\text{st}(r))) = \pi(p, q)^1$.

2.1.3 Diameter, Center, and Diameter Path

For $p, q \in N$, $\text{dist}(p, q)$ is the distance (either the number of edges or the sum of positive edge weights) of the simple path from p to q .

Definition 4 Let $G = (N, E)$ be a connected graph. A simple path $\pi : p, q$ of maximal total distance $\text{dist}(p, q)$ is called a **diameter path**, and $|\pi|$ the **diameter** of G . A node p which minimizes

$$\max_{q \in N} \text{dist}(p, q)$$

is called a **center** of G .

Observation 12 Let $T = (N, E)$ be a tree and let $\pi : m_1 \rightarrow m_2$, $m_1, m_2 \in N$ be a diameter path in T . Then:

- (i) Let $p \in N$ such that π traverses p . Then there exists no $q \in N \setminus \pi$ such that $\text{dist}(p, q) > \min(\text{dist}(p, m_1), \text{dist}(p, m_2))$.
- (ii) π traverses at least one of T 's centers.
- (iii) For all $p \in N$, at least one of the nodes m_1 or m_2 has maximal distance to p .

PROOF: (i) Assume there exists $q \in N \setminus \pi$ with $\text{dist}(p, q) > \min(\text{dist}(p, m_1), \text{dist}(p, m_2))$. Without loss of generality assume further that $\text{dist}(p, m_1) \geq \text{dist}(p, m_2)$. Since the simple paths $\pi(m_1, p)$ and $\pi(p, q)$ have no edge in common, we get

$$\text{dist}(m_1, q) = \text{dist}(m_1, p) + \text{dist}(p, q) > |\pi|, \quad ,$$

which contradicts that π is a diameter path.

(ii) follows directly from (i)

(iii) For $p, q \in N$, choose $i \neq j \in \{1, 2\}$ such that the diameter path from m_i to m_j traverses $\text{MEET}(m_1, m_2, p)$ not before it traverses $\text{MEET}(m_1, m_2, q)$. Without loss of generality we assume $i = 1$.

¹Since the operations HEAD and TAIL destroy the uplinks of the original ET tree ET_i , we have to replace ET_i by ET' .

Further, assume $\text{dist}(q, p) > \text{dist}(m_1, p)$. It follows $\text{dist}(q, r) > \text{dist}(m_1, r)$ for each node $r \in N$ which is traversed by the simple path $\pi(m_1, p)$, especially for $r := \text{MEET}(m_1, m_2, p)$. The choice of i ensures that the simple paths $\pi(q, r)$ and $\pi(r, m_2)$ are edge-disjoint, hence

$$\text{dist}(q, m_2) = \text{dist}(q, r) + \text{dist}(r, m_2) > \text{dist}(m_1, r) + \text{dist}(r, m_2) = |\pi| \quad ,$$

which contradicts that π is a diameter path. \square

Again, let e be the inner node of some ET tree ET_i and $\text{st}(e)$ be the Euler subtour at e . The nodes which are traversed by $\text{st}(e)$ induce a tree $T(e)$ in the forest F , more specifically a subtree of the tree T_i which is represented by ET_i . The main idea is to maintain, for each such e , a diameter path for $T(e)$. Again, we represent diameter paths as persistent arrays. With slight modifications, i.e. storing the edge weights inbetween two adjacent nodes and using an appropriate homomorphism, we can maintain the total cost of each path, which in case of a diameter path yields the diameter of the tree. Furthermore, we can extract tree centers by using a binary search, which needs time $O((\log |\pi|)^2)$ in a persistent array which represents the diameter path π .

The task of maintaining diameter paths in ET trees is complicated by the fact that, for two inner nodes e, e' within the same ET tree, the trees $T(e), T(e')$ are not necessarily disjoint. But the following propositions show that this does no harm:

Observation 13 *Let $T_i = (N_i, E_i)$, $i = 1, 2$ be two (not necessarily edge-disjoint) trees such that $T' = (N_1 \cup N_2, E_1 \cup E_2)$ is also a tree. Then:*

- (i) $(N_1 \cap N_2, E_1 \cap E_2)$ is a tree.
- (ii) A simple path π in T' lies either completely in T_1 or completely in T_2 , or there exist $p \in N_1 \setminus N_2$ and $q \in N_2 \setminus N_1$ such that $\pi : p \rightarrow q$ or $\pi : q \rightarrow p$.

The proof is obvious by using the cycle-freeness property of $(N_1 \cup N_2, E_1 \cup E_2)$.

Proposition 14 *Let $T_i = (N_i, E_i)$, $i = 1, 2$ be two (not necessarily edge-disjoint) trees such that $T' = (N_1 \cup N_2, E_1 \cup E_2)$ is also a tree. Furthermore, choose $m_1, m_2 \in N_1$, $m_3, m_4 \in N_2$ such that $\text{dist}(m_1, m_2)$ is maximal in T_1 and $\text{dist}(m_3, m_4)$ is maximal in T_2 . Then the diameter of T' is*

$$\max_{p, q \in N_1 \cup N_2} \text{dist}(p, q) = \max(\text{dist}(m_1, m_2), \text{dist}(m_1, m_3), \text{dist}(m_1, m_4), \\ \text{dist}(m_2, m_3), \text{dist}(m_2, m_4), \text{dist}(m_3, m_4)) \quad .$$

PROOF: Let p, q be two nodes of maximal distance in the tree T' and $\pi : p = \pi_1, \dots, \pi_n = q$ be a corresponding diameter path. If π lies entirely in T_1 or entirely in T_2 then the diameter of T' is $\text{dist}(m_1, m_2)$ or $\text{dist}(m_3, m_4)$. Otherwise assume without loss of generality that $p \in N_1 \setminus N_2$ and $q \in N_2 \setminus N_1$. Let k be the largest index such that $\pi_1, \pi_2, \dots, \pi_k$ lies entirely in T_1 . According observation 12 (iii), m_1 or m_2 has

maximal distance to r . Hence, in π we can replace $\pi_1, \pi_2, \dots, \pi_k$ by the simple path $\pi(m_i, \pi_k)$ for appropriate $i \in \{1, 2\}$. Analogously, we can replace π_j, \dots, π_n by the simple path $\pi(\pi_j, m_i)$ for appropriate $i \in \{3, 4\}$, where j is the smallest index such that π_j, \dots, π_n lies entirely in T_2 . \square

Given an ET tree ET_i for the tree $T_i = (N_i, E_i)$, we store the following information in each inner node e of ET_i :

- $s(e) = \text{src}(\text{st}(e))$ and $t(e) = \text{dst}(\text{st}(e))$, i.e. the nodes where e 's corresponding Euler subtour starts and ends.
- $m_1(e)$ and $m_2(e)$, which are two nodes of maximal distance in e 's corresponding subtree of $T(e) \subseteq T_i$ (i.e. the tree consisting of all edges which are traversed by e 's Euler subtour).
- The following simple paths: $\pi(s(e), t(e))$, $\pi(m_1(e), m_2(e))$, $\pi(s(e), m_1(e))$, $\pi(s(e), m_2(e))$, $\pi(m_1(e), t(e))$, and $\pi(m_2(e), t(e))$.

Note that $\pi(s(e), t(e))$ has already been used in order to maintain simple paths. The other five paths can be used in order to maintain diameter paths recursively for each inner node e of ET_i , since $\pi(m_1(e), m_2(e))$ is a diameter path of e 's corresponding subtree of $T(e)$:

If e is a leaf node it is clear how to compute the required information. If e has only one child e' , then we take the information from e' . Now assume e, e_1, e_2 are three inner nodes of ET_i such that e_1, e_2 are, respectively, the left and the right child of e . Furthermore, assume we already know $s(e_{1,2}), t(e_{1,2}), \pi(s(e_{1,2}), t(e_{1,2})), \pi(m_1(e_{1,2}), m_2(e_{1,2})), \pi(s(e_{1,2}), m_1(e_{1,2})), \pi(s(e_{1,2}), m_2(e_{1,2})), \pi(m_1(e_{1,2}), t(e_{1,2})),$ and $\pi(m_2(e_{1,2}), t(e_{1,2}))$. By path composition we can compute the simple paths $\pi(s(e), t(e)), \pi(m_1(e), m_1(e_2)), \pi(m_1(e), m_2(e_2)), \pi(m_2(e), m_1(e_2)),$ and $\pi(m_2(e), m_2(e_2))$. By maintaining the total distance of each path, we can use observation 14 in order to choose two nodes $m_1(e), m_2(e) \in \{m_1(e_1), m_2(e_1), m_1(e_2), m_2(e_2)\}$ of maximal distance in $T(e)$. Again, by composition of the appropriate paths, we compute the paths $\pi(m_1(e), m_2(e)), \pi(s(e), m_1(e)), \pi(s(e), m_2(e)), \pi(m_1(e), t(e)),$ and $\pi(m_2(e), t(e))$. The number of compositions is constant, hence for each node e of ET_i we need $O((\log m)^2)$ time, where $m \leq n$ is the maximal number of nodes of any simple path in T_i .

Corollary 15 *In a dynamic forest F , we can, for each maximal tree T_i , maintain diameter, center, and a diameter path in $O(\log n (\log m)^2)$ time per update, where $m \leq n$ is the maximal number of nodes for a simple path in F .*

2.2 The Semiring Framework for Shortest Paths

Shortest-paths and shortest-distance problems on general graphs are often generalized in such a way that they work on semirings. The concept of a semiring abstracts of the operations of adding costs and obtaining minima:

Definition 5 *A semiring is a structure $R = (R, +_R, \times_R, 0_R, 1_R)$ such that*

- (i) $(R, +_R, 0_R)$ is a commutative monoid,
- (ii) $(R, \times_R, 1_R)$ is a monoid,
- (iii) \times_R distributes over $+_R$, that is, for $a, b, c \in R$ the laws of distributivity hold:

$$\begin{aligned} (a +_R b) \times_R c &= (a \times_R c) +_R (b \times_R c) \\ c \times_R (a +_R b) &= (c \times_R a) +_R (c \times_R b) \end{aligned}$$

- (iv) 0_R is an annihilator for \times_R , i.e. for all $a \in R$, $a \times_R 0_R = 0_R \times_R a = 0_R$ holds.

Our prime example is the **tropical semiring** of real numbers $(\mathbb{R}^+ \cup \infty, \min, +, \infty, 0)$ and its relatives which are obtained by restriction to $\mathbb{N} \cup \infty$, $\mathbb{Q}^+ \cup \infty$, or an interval $\{0, \dots, k\} \cup \infty$, $k \in \mathbb{N}$ etc. Another important semiring is, for any set X , the semiring $(2^{X^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$ of languages over X .

2.2.1 All-Pairs-Shortest-Distance

Let $G = (N, E)$ be a complete directed graph — that is, $E = N \times N$ — and $R = (R, +_R, \times_R, 0_R, 1_R)$ be a semiring. Furthermore, let $\delta : E \rightarrow R$ be a mapping of all edges to their respective weights in R . We assume that a path in G is represented as a string of edges $\pi = \pi_1 \cdots \pi_k \in N^*$, $\pi_i \in E$. We can extend δ to paths by defining

$$\delta(\pi) := \prod_{i=1}^k \delta(\pi_i)$$

for paths $\pi = \pi_1 \cdots \pi_k \in E^*$, $\pi_i \in E$. We then call $\delta(\pi)$ the cost or the total distance of the path π . For any two nodes $p, q \in N$, we define their shortest distance $D(p, q)$ as

$$D(p, q) := \sum_{\pi: p \rightarrow q} \delta(\pi) \quad .$$

We call the semiring R **ω -complete** if countably-infinite sums exist in such a way that \times_R distributes over them. This ensures that the sum above exists. The **algebraic all-pairs-shortest-distance problem** is, for given graph $G = (N, E)$, semiring R , and weight-function $\delta : E \rightarrow R$, to compute the $|N| \times |N|$ table D which maps each pair $p, q \in N$ to their shortest distance $D(p, q)$.

Now, if R is ω -complete and if $+_R, \times_R$ are effectively computable, as well as the infinite sum

$$a^* := \sum_{i=0}^{\infty} a^i$$

for each $a \in R$, then the well-known Floyd-Kleene-Warshall algorithm solves the all-pairs-shortest-distance problem. Without loss of generality, we may assume $N = \{0, 1, \dots, n-1\}$. Thus, the input is the $n \times n$ matrix δ with entries $\delta_{ij} = \delta(i, j)$, $i, j \in N$, plus the semiring with its operations $+_R, \times_R, ^*$.

Algorithm 2.3 Floyd-Kleene-Warshall All-Pairs-Shortest-Distance Algorithm

input:

the number of nodes n ,
a $n \times n$ matrix of weights δ for the complete graph with n nodes,
the semiring operations $+_R, \times_R, *$.

output:

the $n \times n$ table D (the transitive closure of δ), containing the shortest distances for each pair of nodes.
 $D^{(0)} := \delta$

for $k := 0$ **to** $n - 1$ **do**
 $D_{ij}^{(k)} := (D_{kk}^{(k-1)})^*$
for $i := 0$ **to** $n - 1$ **do**
for $j := 0$ **to** $n - 1$ **do**
if $i \neq k \vee j \neq k$ **then**
 $D_{ij}^{(k)} := D_{ij}^{(k-1)} +_R D_{ik}^{(k-1)} \times_R D_{kk}^{(k)} \times_R D_{kj}^{(k-1)}$
end if
end for
end for
end for
 $D := D^{(k)}$

Since the result D equals $\sum_{i=0}^{\infty} \delta^i$ (where matrix products are defined in terms of $+_R, \times_R$), we call the shortest-distance matrix D also the **transitive closure** of δ .

2.2.2 Semirings for Paths

We want to show that the Floyd-Kleene-Warshall algorithm can be employed in order to solve the all-pairs-shortest-paths (APSP) problem. Indeed, every algorithm that solves the algebraic all-pairs-shortest-distance for arbitrary ω -complete semirings, can be employed in order to solve the APSP problem. We show this by defining an appropriate semiring, the result then also applies to other algebraic shortest-distance problems, for instance for single-source problems.

Given a complete graph $G = (N, E)$ with $N = \{0, 1, \dots, n-1\}$ and the $n \times n$ matrix of edge weights δ with entries δ_{ij} in the tropical semiring $\mathbb{R}^+ \cup \infty$, we can employ the Floyd-Kleene-Warshall algorithm in order to compute the corresponding minimum **all-pairs-shortest-path** problem. That is, for each $i, j \in N$, we want to find a shortest path $\pi : i \rightarrow j$. In order to employ the Floyd-Kleene-Warshall algorithm, we need to construct an appropriate semiring which encodes operations on paths.

The biggest problem is to choose minima among paths in a way that respects the laws of commutativity and distributivity. We therefore assume that the edges E are equipped with an arbitrary order \leq which induces a lexicographic order on E^* , according to the following definition:

Definition 6 Let $(X, <)$ be some ordered set. The **lexicographic order** $<_l$ on X^* is defined by $v <_l w$ for all $v, w \in X^*$ with $|v| < |w|$ and for all $x, y \in X$ by $x <_l y$ if and only if $x < y$. For all other strings of the form xv, yw with $|v| = |w|$ for $x, y \in X$ and $v, w \in X^*$ we define recursively $xv <_l yw$ if and only if either $x < y$ or if otherwise $x = y$ and $v <_l w$.

For any string $v = v_0 \cdots v_k \in E^*$, $v_i \in E$ we define $\delta(v) := \prod_{i=1}^k \delta(v_i)$. Note that when v happens to be a valid path, then $\delta(v)$ is its cost. We add an absolute maximum value ∞ to the set E^* , that is ∞ is lexicographically larger than any $v \in E^*$ and $\delta(\infty) := \infty$. We define the binary operation $\min : (E^* \cup \infty) \times (E^* \cup \infty) \rightarrow E^* \cup \infty$ as

$$\min(v, w) := \begin{cases} v, & \text{if } \delta(v) < \delta(w) \\ v, & \text{if } \delta(v) = \delta(w) \text{ and } v \text{ is lexicographically strictly smaller than } w \\ w, & \text{else} \end{cases}$$

for $v, w \in E^*$. One can verify that $(E^* \cup \infty, \min, \infty)$ forms a commutative monoid. Furthermore, $(E^* \cup \infty, \min, \cdot, \infty, \varepsilon)$ forms a ω -complete semiring, where $v \cdot w$ is ∞ if $\infty \in \{v, w\}$, and otherwise \cdot is the usual concatenation of strings. Indeed, \cdot is associative with annihilator ∞ and neutral element ε , so it remains to verify the distributive laws. Since both distributive laws can be verified in a similar way, we only show

$$a \cdot \min(b, c) = \min(a \cdot b, a \cdot c), \quad \text{for all } a, b, c \in E^* \cup \infty.$$

Indeed, if $a = \infty$ then both sides will evaluate to ∞ and if $b = \infty$, then both sides evaluate to $a \cdot c$, and similar for $c = \infty$. So let us assume $a, b, c \in E^*$. If $\delta(b) < \delta(c)$, then $\delta(a \cdot b) < \delta(a \cdot c)$, hence both sides evaluate to $a \cdot b$. If $\delta(b) = \delta(c)$ and b is lexicographically smaller than c , then $\delta(a \cdot b) = \delta(a \cdot c)$ and $a \cdot b$ will be lexicographically smaller than $a \cdot c$, hence both sides evaluate to $a \cdot b$. The same if we change the roles of b and c . In the remaining case $b = c$, both sides will evaluate to $a \cdot b$.

Note that the semiring is ω -complete and that for $v \in E^*$ it is $v^* = \min\{\varepsilon, v, vv, \dots\} = \varepsilon$. In order to employ the Floyd-Kleene-Warshall algorithm, we start with the $n \times n$ matrix δ' with the entries $\delta'_{ij} := (i, j)$, that is, each entry δ'_{ij} is the edge from i to j . Note that the semiring $(E^* \cup \infty, \min, \odot, \infty, \varepsilon)$ contains invalid paths. However, if we start the Floyd-Kleene-Warshall algorithm on δ' , it computes for each pair $i, j \in N$:

$$D(i, j) = \min_{\pi_1 \cdots \pi_k : i \rightarrow j} \prod_{i=1}^k \delta_{i_j} = \min_{\pi_1 \cdots \pi_k : i \rightarrow j} \prod_{i=1}^k \pi_i = \min\{\pi : i \rightarrow j\}$$

According to the definition of the operation \min , $D(i, j)$ is the lexicographically first among all minimum paths $\pi : i \rightarrow j$.

If two paths π, π' have the same cost, then choosing the lexicographically first one is an expensive operation. In the worst case, we have to make $O(m)$ pairwise comparisons of edges, where m is the minimum length of π, π' . On the other hand, in many applications it is sufficient to compute one of the

shortest paths for each pair $i, j \in N$, we do not care whether it is the lexicographically first one or not. What we actually want is to identify all paths $\pi : i \rightarrow j$ with equal cost, such that the operation \min can choose in an algorithmically convenient way (even probabilistically) among paths of minimum cost. The easiest way would be to construct a semiring that identifies all paths of equal costs, such that in an implementation, \min chooses among all valid representations. The problem is that our equivalence relation is too coarse: As an example, two paths $\pi : i \rightarrow j$ and $\pi' : i' \rightarrow j'$ might be equivalent with respect to their cost, but still it is $i \neq i'$ or $j \neq j'$. If the equivalence class of the shortest path from i to j is represented by a path $\pi' : i' \rightarrow j'$ with $i \neq i'$ or $j \neq j'$, the result will not be of any use anymore. We can ensure that this does not happen by inspecting the algorithm. In the case of the Floyd-Kleene-Warshall algorithm this is not very difficult, but of course this does not take any advantage of the concept of a semiring as an interface for generic shortest-distance algorithms.

An alternative way is to define a semiring with built-in “type-safety”, which only allows to concatenate or to compute minima of compatible paths and otherwise yields an error-value \perp .

For $i, j \in N$ and $x \in \mathbb{R} \cup \infty$ let

$$[ij; x] := \{\pi : i \rightarrow j : \delta(\pi) = x\}$$

be the set of all paths π from i to j with cost x . We set

$$P(G) := \{[ij; x] : i, j \in N \wedge x \in \mathbb{R} \cup \infty \wedge [ij; x] \neq \emptyset\}$$

and $\Pi(G) := P(G) \cup \{\infty, \varepsilon, \perp\}$. We want to define the two binary operations $\inf, \cdot : \Pi(G) \times \Pi(G) \rightarrow \Pi(G)$ such that $\Pi(G) = (\Pi(G), \inf, \cdot, \infty, \varepsilon)$ is a semiring.

On $\Pi(G)$, we define the following partial order $<$: \perp and ∞ are absolute minimum and maximum, that is $\perp \leq a \leq \infty$ for all $a \in \Pi(G)$. For ε , we define $[ii; 0] < \varepsilon$ for all $i \in N$. Finally, for all $i, j \in N$, we define $[ij; x] < [ij; y]$ if and only if $x < y$. All other pairs of values from $\Pi(G)$ are incomparable by $<$.

For $a, b \in \Pi(G)$, we let $\inf(a, b)$ take the infimum of a, b with respect to the previously defined order $<$ on $\Pi(G)$. One can show that for our specific $<$, such an infimum exists and that it is unique. Then $(\Pi(G), \inf, \infty)$ is a commutative monoid with ∞ being its neutral element.

For $a, b \in \Pi(G)$, we now define $a \cdot b$: If $\infty \in \{a, b\}$, we set $a \cdot b := \infty$, hence ∞ annihilates. For all other values, \perp annihilates, that is $\perp \cdot a = a \cdot \perp = \perp$ whenever $a \neq \infty$. Furthermore, ε acts like a neutral element, i.e. $\varepsilon \cdot a = a \cdot \varepsilon = a$ for all $a \in \Pi(G)$. In the remaining cases assume $a = [i_a j_a; x] \in P(G)$ and $b = [i_b j_b; y] \in P(G)$ for $i_a, j_a, i_b, j_b \in N, x, y \in \mathbb{R}^+ \cup \infty$. Then

$$[i_a j_a; x] \cdot [i_b j_b; y] := \begin{cases} [i_a j_b; x + y], & \text{if } j_a = i_b \\ \perp, & \text{else} \end{cases}$$

The result of $[i_a j_a; x] \cdot [i_b j_b; y]$ is again in $\Pi(G)$: If there exist valid paths $\pi : i_a \rightarrow j_a$ and $\pi' : i_b \rightarrow j_b$ with costs x, y respectively, and if $j_a = i_b$, then $\pi \cdot \pi'$ is a valid path with cost $x + y$, hence $[i_a j_b; x + y]$ is

a non-empty set and therefore in $P(G)$.

By definition, ε is a neutral element for \cdot . Also, \cdot is associative, that is $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for all $a, b, c \in \Pi(G)$: If $\infty \in \{a, b, c\}$ it annihilates by definition and both sides evaluate to ∞ . If otherwise $\perp \in \{a, b, c\}$ and $a, b, c \neq \infty$, then \perp annihilates and both sides evaluate to \perp . Associativity is easy to verify if $\varepsilon \in \{a, b, c\}$. If now $a = [i_a j_a; x], b = [i_b j_b; y], c = [i_c j_c; z] \in P(G)$, then both sides evaluate to $[i_a j_c; x + y + z]$ if and only if $j_a = i_b$ and $j_b = i_c$, otherwise they evaluate to \perp .

In order to show that $\Pi(G) = (\Pi(G), \inf, \cdot, \infty, \varepsilon)$ is a semiring, it remains to show that the two laws of distributivity hold. Since both laws can be verified in essentially the same way, we will only show

$$a \cdot \inf(b, c) = \inf(a \cdot b, a \cdot c), \quad \text{for all } a, b, c \in \Pi(G) \quad .$$

First we assume $\infty \in \{a, b, c\}$. If $a = \infty$, both sides evaluate to ∞ . If $b = \infty$, then both sides evaluate to $a \cdot c$ because of $a \cdot \infty = \infty$. The same with $c = \infty$.

If $a, b, c \neq \infty$ but $a = \perp$, then both sides evaluate to ∞ if $b = c = \infty$, and to \perp otherwise. If $a, b, c \neq \infty$ and $b = \perp$ or $c = \perp$, both sides evaluate to \perp . If $a = \varepsilon$ then both sides evaluate to $\inf(b, c)$. If $a = [i_a j_a; x]$ for $i_a, j_a \in N$ and $x \in \mathbb{R}^+ \cup \infty$ and $b = \varepsilon$ we distinguish the following cases: First, if $c = \varepsilon$ then both sides will evaluate to a . Secondly, if $c = [i_c j_c; z]$ for $i_c, j_c \in N$ and $z \in \mathbb{R}^+ \cup \infty$, then if $j_a \neq i_c$ we get \perp on both sides. If on the other hand $j_a = i_c$, then we get a on both sides if and only if $i_c = j_c$, otherwise we get \perp . It remains to show distributivity for $a, b, c \in P(G)$. So assume that $a = [i_a j_a; x], b = [i_b j_b; y], c = [i_c j_c; z] \in P(G)$. Under this assumption, we observe that none of the subterms of $a \cdot \inf(b, c)$ and $\inf(a \cdot b, a \cdot c)$ can evaluate to ∞ , hence \perp will annihilate whenever it occurs during evaluation. Indeed, if $j_a = i_b = i_c$ and $j_b = j_c$, then both sides evaluate to $[i_a j_b, x + \inf y, z]$, otherwise they evaluate to \perp .

Finally, one can verify that the semiring $\Pi(G) = (\Pi(G), \inf, \cdot, \infty, \varepsilon)$ is ω -complete, and for $[i_a i_b; x] \in P(G)$ it is

$$[i_a i_b; x]^* = \begin{cases} \varepsilon, & \text{if } i_a = i_b \\ \perp, & \text{else} \end{cases},$$

and $\infty^* = \varepsilon^* = \varepsilon$ and $\perp^* = \perp$.

As we have seen, $\Pi(G) = (\Pi(G), \inf, \cdot, \infty, \varepsilon)$ is a semiring. Each $[ij : x] \in P(G)$ can be represented by a path $\pi : i \rightarrow j$ with cost x , in form of a persistent array plus a number $x \in \mathbb{R}^+ \cup \infty$. This permits efficient implementation of the operations \cdot and \inf . The Floyd-Kleene-Warshall algorithm, operating on $\Pi(G)$, will then yield a representation for one of the shortest paths for all pairs $i, j \in N$. In order to employ the Floyd-Kleene-Warshall algorithm, we start with the $n \times n$ matrix δ' with the entries

$\delta'_{ij} := [ij; \delta(i, j)]$, where $[ij; \delta(i, j)]$ is represented by the single edge (i, j) of cost $\delta(i, j)$. If we start the Floyd-Kleene-Warshall algorithm on δ' , it computes for each pair $i, j \in N$:

$$\begin{aligned} D(i, j) &= \inf_{\pi_1 \dots \pi_k : i \rightarrow j} \prod_{i=1}^k [\pi_i; \delta(\pi_i)] \\ &= \inf_{\pi : i \rightarrow j} [ij; \delta(\pi)] \\ &= [ij; \inf\{\delta(\pi) | \pi : i \rightarrow j\}] = [ij; \min\{\delta(\pi) | \pi : i \rightarrow j\}] \end{aligned}$$

Since in our implementation $[ij; \min\{\delta(\pi) | \pi : i \rightarrow j\}]$ is represented as a valid path $\pi : i \rightarrow j$ with cost $\min\{\delta(\pi) | \pi : i \rightarrow j\}$, the matrix D gives us a solution of the APSP problem for G, δ .

2.3 All Pairs-Shortest-Paths and Successor Matrices

We have seen how persistent arrays under concatenation can be fitted into the semiring framework of Floyd-Kleene-Warshall's all-pairs-shortest-path (APSP) algorithm. The Floyd-Kleene-Warshall algorithm will compute a matrix of shortest paths by using $O(n^3)$ array concatenations. In some cases, there exist faster algorithms. For instance, R. Seidel's algorithm [Sei 95], which is originated in preceding work of Alon, Galil, and Margalit, reduces the APSP problem for undirected unweighted graphs to boolean matrix multiplication. The fastest known algorithm for multiplying two $n \times n$ boolean matrices is due to Coppersmith and Winograd and runs in $O(n^{2.376})$ time (see [CoWi 90]) while Seidel's algorithm introduces an additional factor of $O((\log n)^2)$. We will not discuss Seidel's algorithm in this thesis, a good explanation of the algorithm can be found in [MoRa 95]. The result of an APSP instance is commonly represented as successor matrix, permitting reconstruction of the shortest paths.

The purpose of this section is to show that there is no need for the effort of adapting the inner loops of Seidel's algorithm in order to represent the result as a matrix of persistent arrays. Instead, we will show how any $n \times n$ successor matrix can be transformed into a matrix of persistent arrays in $O(n^2 \log n)$ time.

Definition 7 Let $G = (N, E)$ be a graph. Assume the set of nodes is numbered, i.e. $N = \{0, 1, \dots, n-1\}$. A successor matrix for G is a $n \times n$ matrix R with entries in $\infty, 0, 1..n-1$ such that for $0 \leq i, j < n$, if $i = j$ then $S_{ij} = j$. For $i \neq j$ it is $S_{ij} = \infty$ if and only if $i, j \in N$ are disconnected and $S_{ij} \in N$ if and only if there exists a simple path $\pi : i \rightarrow j$ such that π traverses the node S_{ij} directly after i .

In this section, paths are again represented as sequences of edges. In order to obtain a path from i to j , and assuming that i, j are connected, we only need to go from i to S_{ij} and then continue successively until we reach j . Indeed, for any instance of the APSP problem there exists a solution which can be represented as a successor matrix (but not every solution).

The task is to transform a given $n \times n$ successor matrix R in such a way that we get a $n \times n$ matrix P with entries in $N^* \cup \infty$ such that for $i, j \in N$, $P_{ij} = \infty$ if and only if i, j are disconnected, else P_{ij} is a shortest path $\pi : i \rightarrow j$.

The idea is to compute matrices $S^{(k)}$, $k = 0..[\log n]$, starting with $S^{(0)} = S$. For disconnected $i, j \in N$, all

$S_{ij}^{(k)}$, $k = 0.. \lceil \log n \rceil$, are ∞ . Now assume that $i, j \in N$ are connected and let $\pi = \pi_0 \dots \pi_m : i \rightarrow j$ be the path from i to j which is described by the successor matrix R . Then the entry $S_{ij}^{(k)}$ of matrix k is defined to be the 2^k th node π_{2^k} of the path π if $2^k \leq m$, and otherwise $S_{ij}^{(k)} = j$. At the same time, we maintain a $n \times n$ matrix $P^{(k)}$, such that if i, j are connected, then $P_{ij}^{(k)} \in N^*$ is a shortest path from i to $S_{ij}^{(k)}$. Note that a shortest path which is described by a successor matrix can never traverse more than n edges, hence after $k \leq \log n$ iterations we have that $P_{ij}^{(k)}$ is a shortest path from i to $S_{ij}^{(k)} = j$.

Algorithm 2.4 Transforming a successor matrix R into a matrix P of paths

```

 $S^{(0)} := R$ 
for  $i := 0$  to  $n - 1$  do
    for  $j := 0$  to  $n - 1$  do
        if  $i = j$  then
             $P_{ii}^{(0)} := \varepsilon$ 
        else if  $S_{ij} = \infty$  then
             $P_{ij}^{(0)} := \infty$ 
        else
             $P_{ij}^{(0)} := [i, S_{ij}]$ 
        end if
    end for
end for
for  $k := 1$  to  $\lceil \log n \rceil$  do
     $P^{(k)} := P^{(k-1)}$ 
     $S^{(k)} := S^{(k-1)}$ 
    for  $i := 0$  to  $n - 1$  do
        for  $j := 0$  to  $n - 1$  do
            if  $i \neq j \wedge S_{ij}^{(k-1)} \neq \infty \wedge S_{ij}^{(k-1)} \neq j$  then
                 $R := S_{ij}^{(k-1)}$ 
                 $P_{ij}^{(k)} := P_{ij}^{(k-1)} \cdot P_{sj}^{(k-1)}$ 
                 $S_{ij}^{(k)} := S_{sj}^{(k-1)}$ 
            end if
        end for
    end for
end for
 $P := P^{(\lceil \log n \rceil)}$ 
    
```

$P^{(0)}$ can be computed in $O(n^2)$ time, and since each iteration needs n^2 many concatenations of persistent arrays, the total running time is $O(n^2(\log n)^2)$. A more detailed analysis reveals that for each pair i, j , all except one of the $\lceil \log n \rceil$ many concatenations either involve an empty array or two arrays represented by trees of equal height (if we use our implementation of persistent arrays as AVL trees).

Those are special cases of concatenation which can be handled in $O(1)$ time. So finally, the time bound for our algorithm is actually $O(n^2 \log n)$.

The space requirements are bounded by the execution time of the algorithm, hence the resulting matrix P requires $O(n^2 \log n)$ space.

Corollary 16 *Given a $n \times n$ successor matrix S for a graph $G = (N, E)$ and a monoid homomorphism $h : E^* \rightarrow M$, it is possible to compute $h(\pi)$ simultaneously for all shortest paths π which are represented by S . The computation requires $O(n^2 \log n)$ time plus the time needed for $O(n^2 \log n)$ multiplications in M and for computing $O(|E|)$ images $h(x)$, $x \in E$.*

PROOF: In algorithm 2.4, we can maintain the images $h(\pi)$ of paths. The additional costs involved are exactly those stated in the theorem. Another approach would be to modify algorithm 2.4 in such way that it computes the images $h(\pi)$ directly. \square

2.4 Related Work

The main contribution of the first section of this chapter is a method for maintaining path information within ET trees. ET-trees have been introduced by Henzinger and King, see [HeKi 99]. Another data structure for dynamic trees are topology trees, which have been proposed by Fredrickson, see [Fre 85]. The article [AHLT 03] by Alstrup, Holm, de Lichtenberg, and Thorup gives an alternative description called top trees. There the authors show how to maintain diameter, center and several other tree parameters. The article [EGI 99] by Eppstein, Galil, and Italiano gives a extensive overview on dynamic graph algorithms. It should be noted that our implementation of dynamic trees, which uses ET trees, does not fully implement Sleator and Tarjan's axiomatic interface for dynamic trees, since it does not allow efficient updates of edge-weights along a path. On the other hand, ET trees have been used by Holm, de Lichtenberg, and Thorup ([HLT 01]) for dynamic connectivity problems on general undirected graphs, where the total costs per update are $O((\log n)^2)$. It is reasonable to assume that the additional overhead for maintaining path information by persistent arrays does not harm much in those cases.

The observation 14 is a generalization of equivalent results for disjoint trees.

The semiring framework for Floyd-Kleene-Warshall's algorithm can be found in many textbooks, see for instance [AHU 74] and the first edition of [CLRS 01]. The article by M. Mohri [Moh 02] shows several semirings in context of shortest-path and shortest-distance problems. The construction of a semiring for paths represented as persistent arrays is by the author of this thesis.

Many APSP problems represent their solutions as successor — or alternatively predecessor — matrices. The algorithm 2.4 for transforming a successor matrix to a matrix that represents paths as persistent arrays is new in this thesis. Remarkable is the small additional space overhead, since persistent arrays are much more versatile for subsequent processing of the result.

Chapter 3

Homomorphisms on Arrays

A compact representation of large arrays in memory is not very useful if we cannot efficiently extract some information from it. The homomorphism technique provides means for computing with exponentially large, persistent, arrays. There are many functions on arrays which can be considered as homomorphic images, or where homomorphic images help during the course of computation. In particular we provide probabilistic polynomial-time algorithms for equality test and lexicographic comparison. We also provide deterministic polynomial-time algorithms for sorting elements and for the generalized state machine.

The homomorphism technique for arrays is also useful for algorithm design. This has already been employed in the field of parallel algorithms, in computer algebra, and many other fields of algorithm design. We give some further examples in this chapter, in particular interactive line breaking.

3.1 Algorithms on Large Arrays

3.1.1 Polynomial Evaluation and Inequality Tests

Let $R = (R, +, \cdot, 0, 1)$ be a commutative semiring. By $R[z]$ we denote the semiring of single-variabled polynomials over R . We wish to evaluate a given polynomial $f(z) = a_n z^n + a_{n-1} z^{n-1} + \dots + a_0 \in R[z]$ of degree n for some fixed value x of the variable z , *adaptively to changing coefficients* a_0, \dots, a_n :

Let $A \in R^*$ be some array of length $|A| = n + 1$ with elements $A[i] = a_i$, $i = 0..n$, from R . The corresponding polynomial of degree $d_A = n$ is the polynomial f_A with a_i as the coefficient for the i th power of its argument. For some fixed $x \in R$, we want to evaluate

$$f_A(x) := a_n x^n + a_{n-1} x^{n-1} + \dots + a_0 \quad .$$

On A we allow update operations such as changing one of the coefficients a_i , $i = 0..n$. During updates, we wish to maintain the polynomial's value at x : If A' is the array that results from A by changing a_i to a'_i , we want to get immediatly the new value for $f_{A'}(x)$. If R is a ring, the new value can be computed by

$$f_{A'}(x) = f_A(x) + (a'_i - a_i)x^i \quad .$$

Sometimes this is not practical, for instance if the computer cannot perform arithmetic operations in R without round-off errors. Also it might be that R is a semiring which lacks additive inverses.

In some applications it can also be useful to maintain polynom values for more complex updates, such as concatenation and splitting of the arrays of coefficients. An example is the equality test which we discuss later in this section.

For a fixed value $x \in R$, consider the monoid $R_x = (R \times R, \cdot_x, 1_x)$ with $1_x := (0, 1)$ and

$$(q, r) \cdot_x (q', r') := (q + rq', rr') \quad ,$$

for all $q, q', r, r' \in R$. One can verify that R_x is a monoid. We define $h_x : R^* \rightarrow R_x$ as the map which sends each array $A \in R^*$ to $(f_A(x), x^{|A|})$. Then h_x is a monoid homomorphism from R^* to R_x :

$$h_x(AB) = (f_{AB}(x), x^{|AB|}) = (f_A(x)x^{|A|}f_B(x), x^{|A|}x^{|B|}) = h_x(A) \cdot_x h_x(B) \quad (A, B \in R^*)$$

It is the homomorphism on A^* which is (uniquely) defined by $h([a]) := (a, x)$ for all $a \in R$. Together with the discussion from section 1.4.3 we get:

Proposition 17 *For a given value $x \in R$, a persistent array A can be maintained which holds coefficients $A[i] = a_i \in R$ as well as the value $f_A(x)$ such that the following updates are allowed: changing of a single coefficient, and getting new polynomials by concatenating or splitting arrays. All updates involve at most $O((\log |A|)^2)$ semiring operations.*

Furthermore, given a value $x \in R$ and a persistent array $A \in R^$ represented as a binary tree which occupies k nodes in memory, the value $p_A(x)$ can be computed with $O(k)$ semiring operations.*

We want to employ proposition 17 for testing if two given arrays are equal. As a start, we want to explain what the problem: Referring to figure 3.1, let A be the persistent array which is represented by

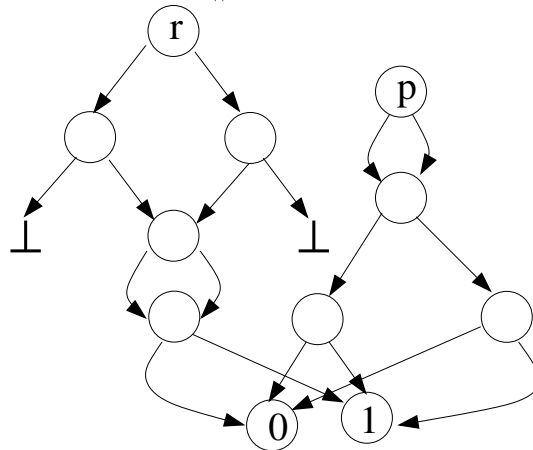


Figure 3.1: The two nodes r and q both represent the same string 01010101. (pointers to the symbol \perp are void)

the root r , and let B be the array represented by the root p . In both cases, the corresponding sequence of

elements is $\text{seq}(A) = \text{seq}(B) = 01010101$, hence we would consider A and B being equal. When using persistent arrays, we do almost never care about their actual representation in memory. Generally we say that two persistent arrays are equal, whenever the corresponding sequence of elements are equal. The problem of deciding equality arises with very large arrays, in particular those which's length grows exponentially in the number of nodes. In this case, iterating through the sequence of elements and making a one-by-one comparison is unfeasible.

There exists a common technique which helps us to provide a probabilistic (in)equality test: Given two arrays $A = [a_0, \dots, a_{n-1}]$, $B = [b_0, \dots, b_{n-1}]$ of equal length n and an errorbound $\varepsilon > 0$, we choose a random number p . Then we compute fingerprints of both arrays modulo p in such a way that if A and B are equal, the fingerprints always coincide. If A and B are unequal, the probability that the fingerprints are different is at least $1 - \varepsilon$.

The resulting algorithm is of onesided-error Monte-Carlo type, i.e. it always terminates and for equal A, B , it always yields the correct answer by comparing both fingerprints. Termination is guaranteed in time polynomial in the size of the representation of A, B as binary trees and the number of leading zeros in the binary (fixed-point) representation of ε .

When dealing with probabilistic algorithms, the underlying machine model is a RAM with a sequence of uniformly distributed independent random bits $X_0 X_1 X_2 \dots$. This allows us (in many practical cases) to consider all results during the computation as random variables over the discrete probability space which corresponds to the atomic events X_0, X_1, \dots, X_k , where k is the maximum number of used random bits on the given input. Note that for Las-Vegas type algorithms, the model is slightly more involved since the number of random bits can be unbounded. For more details about discrete probability and probabilistic algorithms, see the [MoRa 95].

Assume that the number of distinct elements of A and B is at most m , i.e.

$$m := |\{a_i, b_i : i = 0..n-1\}|.$$

It is easy to deduce m from the binary tree representation of A, B by traversing its leave nodes. We can then replace each element from A, B by a unique identifier from the set of numbers $0..m-1$. Without loss of generality, from now on we assume that all elements are numbers between 0 and $m-1$. Now A and B are equal if and only if the corresponding polynomials f_A, f_B are equal at point m :

$$A = B \quad \Leftrightarrow \quad f_A(m) = f_B(m)$$

The reason is that the value $f_A(m) = \sum_{i=0}^{n-1} a_i m^i$ is the number which $A = [a_0, \dots, a_{n-1}]$ represents in m -adic representation, which is — for fixed length n — unique. The same for B and $f_B(m)$. Proposition 17 ensures that the values $f_A(m), f_B(m)$ can be computed with a number of operations which grows polynomially in n . The only problem is that the number of bits needed can be as much as $O(n)$, which can be exponential in the size of A and B 's representation. Instead of computing $f_A(m), f_B(m)$ over the

integers, we perform all computations modulo a random number p . When the number of bits for p is polynomially bounded, we can compute the fingerprints $f_{A,p} := f_A(m) \bmod p$ and $f_{B,p} := f_B(m) \bmod p$ polynomial time. We then have

$$A = B \quad \Rightarrow \quad f_{A,p} = f_{B,p}.$$

The other direction is, of course, not true in general. However, if $f_A(m) = f_B(m)$ modulo p , then p divides $|f_A(m) - f_B(m)|$, which has at most

$$\log_2 |f_A(m) - f_B(m)| \leq \log_2(m^{n+1}) \leq n'$$

prime divisors for $n' := (n+1) \log_2 m$. Hence if p happens to be a prime number, then the fingerprints $f_{A,p}, f_{B,p}$ can only coincide if p is among the prime divisors of $|f_A(m) - f_B(m)|$, from which there are at most n' many. The trick is to choose p as a prime number uniformly from an interval $0.. \tau$ where τ is large enough such that the probability that p is among the prime divisors of $|f_A(m) - f_B(m)|$ is negligible.

The prime number theorem ensures that there exist enough primes already for small k : For an integer x , denote the number of primes smaller or equal to x by $\pi(x)$. We will discuss its application in such a way that an implementation with guaranteed errorbounds is possible. We use Rosser and Schoenfeld's inequality

$$\ln x - \frac{3}{2} < \frac{x}{\pi(x)} < \ln x - \frac{1}{2},$$

from which

$$\frac{x}{2 \ln x} \leq \pi(x)$$

follows. We then choose $\tau \geq 5n'\epsilon^{-1} \ln(n'\epsilon^{-1})$. The number of bits for the binary representation of τ depends only polynomially on n (since $n' = (n+1) \log_2 m$ and $m \leq n$) and the number of leading zeros in ϵ 's fixed-point representation. If $n' \leq 10$ we can decide equality by comparing the array elements one-by-one. If $\epsilon > 0.1$ it does not do any harm to test equality for a better errorbound. Hence without loss of generality, we can assume $n' \geq 10$ and $\epsilon \leq 0.1$ and obtain

$$\begin{aligned} \text{Prob}(f_A = f_B \mid A \neq B \wedge p \text{ is prime}) &= \frac{n'}{\pi(\tau)} \\ &\leq \frac{2n' \ln \tau}{\tau} \\ &\leq \frac{2n' \ln(5n'\epsilon^{-1} \ln(n'\epsilon^{-1}))}{5n'\epsilon^{-1} \ln(n'\epsilon^{-1})} \\ &\leq \frac{2n' \ln(6n'\epsilon^{-1})}{5n'\epsilon^{-1} \ln(n'\epsilon^{-1})} \\ &= \frac{2n'(\ln 6 + \ln(n'\epsilon^{-1}))}{5n'\epsilon^{-1} \ln(n'\epsilon^{-1})} \\ &\leq \frac{2n'(2 \ln(n'\epsilon^{-1}))}{5n'\epsilon^{-1} \ln(n'\epsilon^{-1})} \\ &= \frac{4}{5} \epsilon \leq \epsilon \end{aligned}$$

when f_A, f_B are interpreted as random variables¹. There exist polynomial-time probabilistic algorithms which generate, on input $k \in \mathbb{N}$, and errorbound $\varepsilon > 0$ a prime number candidate p with k bits, in the following sense: For the returned candidate p (a random variable which depends on the machine's random bits)

$$\text{Prob}(p \text{ is not prime}) \leq \varepsilon ,$$

and $\text{Prob}(p = p_1) = \text{Prob}(p = p_2)$ for all prime numbers $p_1, p_2 \leq 2^k$. See for instance [MoRa 95]. Combining such an algorithm with the computation of the fingerprints f_A, f_B yields an errorbound

$$\text{Prob}(f_A = f_B \mid A \neq B) \leq \text{Prob}(f_A = f_B \mid A \neq B \wedge p \text{ is prime}) + \text{Prob}(p \text{ is not prime}) \leq 2\varepsilon .$$

In practice, software libraries often provide a candidate-tester pair of Monte-Carlo type functions for prime number generation for with a built-in errorbound $\alpha > 0$. That is, the probabilistic function `CANDIDATEPRIME(k)` yields a number with k bits which is prime with probability at least $1 - \alpha$. The probabilistic function `TESTPRIMALITY(p)` decides whether $p \in \mathbb{N}$ is a prime. The primality test may yield the wrong answer with probability $\leq \alpha$ but is always "yes" if p is prime. In order to obtain a smaller errorbound $\varepsilon \leq \alpha$, one can still use such a candidate-tester pair: We generate up to $r \geq \log_\alpha(\varepsilon/2)$ prime candidates, each one with errorbound α , from which we choose the first one which passes r primality tests (with errorbound α). If no candidate passes its r primality tests, we return any number we like. The probability that the chosen candidate is a prime number is then at least $1 - \varepsilon$.

3.1.2 Lexicographic Comparison

The probabilistic equality test can be adapted to a probabilistic lexicographic comparison. The main problem is, for two unequal arrays $A = [a_0, \dots, a_{n-1}]$, $B = [b_0, \dots, b_{n-1}]$ to find the first index i such that $a_i \neq b_i$. This is the same as finding the longest equal prefixes $[a_0, \dots, a_{i-1}] = [b_0, \dots, b_{i-1}]$. We can find i by recursively splitting the search interval $0..n-1$ into two parts $0..i-1$ and $i..n-1$ of approximately the same length and continue our search in the first of the two intervals for which the values of A and B differ. This binary search will take time which is polynomial in the size of the array's binary searchtree representation. By calling each equality test with errorbound ε/k , where $k = \lceil \log n \rceil$ is the maximum number of comparisons, the resulting errorbound will be less than the probability of error for at least one of the k tests, which then is $\leq \varepsilon$.

3.1.3 Equality Tests for Arrays which May Contain Arrays

The probabilistic equality test can be generalized to an equality test for composite structures based on persistent arrays. Until now we usually considered sequences over some ground set X . However, when we use persistent arrays as a basic type of a programming language, there is little reason not to allow arrays themselves to be elements of other arrays.

¹In the literature it is common to avoid the formal details when switching from some object to its interpretation as a random variable.

Let X be a finite set of **atoms**. (For instance the set of all instances of primitive types which fit into a register). The **universe over** X is the set $\mathcal{U}(X)$ which is recursively defined as:

$$\mathcal{U}_0(X) := X$$

$$\mathcal{U}_{i+1}(X) := (\bigcup_{j=0..i} \mathcal{U}_j(X))^*$$

$$\mathcal{U}(X) := \bigcup_{i=0,1,2,\dots} \mathcal{U}_i(X)$$

If $A, B \in \mathcal{U}(X)$ they can be transformed into arrays $A', B' \in X \cup \{(\cdot, \cdot)\}^*$ in polynomial time such that A equals B if and only if A' equals B' . We assume $(\cdot, \cdot) \notin X^*$ and flatten an array $A \in \mathcal{U}(X)$ to an array $A' \in X^*$ by recursively replacing each non-atom $[x_1, x_2, \dots, x_n]$ by the sequence $(x_1 x_2 \dots x_n)$. For an binary tree representation, this can be achieved in polynomial time by traversing the tree.

3.1.4 Sorting of Elements

Let $A = [a_0, \dots, a_{n-1}]$ be an array of elements $a_i \in X$, where X is linearly ordered by the relation \leq . Our goal is to sort A , i.e. to create an array $A' = [a'_0, \dots, a'_{n-1}]$ with

$$a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$$

such that for each $x \in X$, its number of occurrences in A equals its number of occurrences in A' . This can be achieved by the following algorithm: First, we compute the set

$$X_A := \{a_i : i = 0..n-1\}$$

of all elements of A . This can be done by traversing the tree which represents A or, alternatively, by the applying the homomorphism which maps an array to the set of its elements. In the next step, we compute for each $x \in X_A$ its number of occurrences in A by computing the image $h_x(A)$ for the homomorphism $h_x : X^* \rightarrow (\mathbb{N}, +, 0)$ defined by

$$h_x(y) \begin{cases} 1, & \text{if } x = y \\ 0, & \text{else} \end{cases} \quad (y \in X).$$

Now, for each element $x \in X_A$, we create an array representing the sequence $x^{h_x(A)}$ by fast exponentiation, and then we concatenate these arrays in the right order. Note that actually we only need to sort the set X_A of all elements of A , which has at most the size of the binary tree which represents A .

3.1.5 Generalized State Machine and Acceptor

Let X be some finite alphabet. A **generalized state machine** M which transforms strings from X^+ to strings from X^* is a tuple $M = (X, S, s_0, \delta_1, \delta_2)$. Here, S is a finite set of states, s_0 is the initial state, and $\delta_1 : S \times X \rightarrow S$, $\delta_2 : S \times X \rightarrow X^*$ constitute M 's program and describe the transducer's behavior on an input string $u \in X^+$: We can extend $\delta_{1,2}$ on $S \times X^+$ by recursively defining

$$\delta_1(s, ux) := \delta_1(\delta_1(s, u), x)$$

and

$$\delta_2(s, ux) := \delta_2(s, u)\delta_2(\delta_1(s, u), x)$$

for $u \in X^+$, $x \in X$ and $s \in S$. For an input $u \in X^+$, we set $M(u) := \delta_2(s_0, u)$.

A generalized state machine $M = (X, S, s_0, \delta)$ can also be defined in terms of a homomorphism $h_M : X^* \rightarrow S^S \times (X^*)^S$. Multiplication in $S^S \times (X^*)^S$ is defined by

$$((f_1, f_2) \cdot (g_1, g_2))(s) := (g_1(f_1(s)), f_2(s)g_2(f_1(s))) \quad (s \in S)$$

for $f_1, g_1 \in S^S$ and $f_2, g_2 \in (X^*)^S$. The neutral element is $(s \mapsto s, s \mapsto \varepsilon)$. The homomorphism h_M is defined by

$$h_M(x) := (s \mapsto \delta_1(s, x), s \mapsto \delta_2(s, x)) \quad (x \in X).$$

Now, for a string $u \in X^+$, we get

$$h_M(u) = (s \mapsto \delta_1(s, u), s \mapsto \delta_2(s, u)),$$

hence $(h_M(u))(s_0) = (\delta_1(s_0, u), M(u))$. The elements from the monoid $S^S \times (X^*)^S$ can be represented in size polynomial in $|S| \cdot \max_{s \in S, x \in X} \delta_2(s, x)$, which is a natural measure for the size of M . Hence the multiplication as defined above can be performed in polynomial time. Together with the results from section 1.4.3, we get a polynomial time algorithm for computing $M(u)$ when u is the sequence of elements of a persistent array represented by a binary tree. Note that in an implementation, it will be wise not to follow the homomorphism technique too doggedly. It is generally better to traverse left subtrees first in the representation of the array and only compute transitions which really occur when processing the input string.

A **deterministic finite acceptor** M can be considered as a special generalized state machine, which accepts an input string $u \in X^+$ if and only if $M(u)$ ends with “yes” (symbolizing and accepting state). Only slight modifications will allow us to search for patterns described by a deterministic finite automaton in polynomial time.

3.2 Algorithm Design

In this section we want to give two further examples for the use of homomorphisms or binary trees for on arrays as a tool in algorithm design. Since the idea has been employed in many areas of algorithm design since long, we only introduce two applications which evolved during our own research.

3.2.1 Poisson Trials

A Poisson trial with success probability p is a random experiment with only two possible outcomes: 0 with probability $1 - p$ and 1 with probability p . If a Poisson trial with fixed success probability is

repeated independently, it is called a Bernoulli trial. An application of a Bernoulli trial is for instance the simulation of a system where different events might occur any time with constant probability. More formally, let the random variables X_i , $0 \leq i < n$ describe n independent Poisson trials, each one with success probability p_i . We want to generate a random sample and collect those trials with positive outcome. In other words, the task is to pick each item $i \in \{0, 1, \dots, n-1\}$ with probability p_i independently and then put the chosen items into a set L . The generic algorithm is algorithm 3.2.1.

Algorithm 3.2.1 *The generic algorithm for drawing a sample of n independent trials:*

```

 $L := \emptyset$ 
for  $i := 0$  to  $n - 1$ 
   $r := \text{RANDOM}[0, 1]$ 
  if  $r \leq p_i$  then  $L := L \cup \{i\}$ 
endfor

```

We assume that $\text{RANDOM}[0, 1]$ generates uniformly and with high enough precision a random floating point number in the interval $[0..1]$.

Algorithm 3.2.1 typically occurs implicitly in simulations of stochastic processes and also in some randomized algorithms. Often the algorithm needs to be repeated many times, so each Poisson trial X_i becomes a Bernoulli trial. If the p_i are much smaller than $1/2$, iterating through all p_i for only a small amount of items that are actually chosen becomes inefficient — we generate a low-entropy sample with a large number of random bits.

For example, when simulating system in a critical environment, such as computers with faulty memory cells, algorithm 3.2.1 becomes very time-consuming if the p_i are small: it generates $O(n)$ random numbers for only a small amount of positive outcomes. If the drawing of the samples is repeated many times, it might be better to use analytic techniques in order to generate samples with positive outcome for each X_i ahead of time and to queue them up until needed.² This might be the fastest way if we draw samples unconditionally, but it does not allow to draw samples with a fixed number of positive outcomes in order to study the system under extremal conditions. We could go back to algorithm 3.2.1 and repeat it until the drawn sample has the required number of positive outcomes, however this turns out to be unfeasible if the p_i are small.

The algorithm that we describe here makes better use of the random bits than algorithm 3.2.1 does and allows to draw samples according to a required number of positive outcomes: Let the X_i , $0 \leq i < n$, be our n Poisson trials, i.e. independent random variables where $X_i = 0$ with probability $1 - p_i$ and $X_i = 1$ with probability p_i . Our algorithm first generates a random number e with the distribution of $Y =$

²For the analytic properties of Bernoulli trials see [Knu 98]

$X_0 + \dots + X_{n-1}$. Then it picks exactly e items from the set $\{0..n-1\}$ according to their probabilities. In order to facilitate this, the algorithm makes use of a recursive splitting of Y which has to be computed in a preprocessing stage. We describe both, the preprocessing and the algorithm for choosing the numbers, in the next section.

It is well-known that the distribution of $Y = X_0 + \dots + X_{n-1}$ is the n -fold convolution of the probabilities p_i and therefore can be computed with $O(n(\log n)^2)$ floating point operations by using fast Fourier transformation (FFT) for the convolution products. The consideration of floating point precision and roundoff errors is application-specific. However, depending on the randomized algorithm, we might want to bound the required floating point precision as well as the number of random bits that are needed. In stochastic simulations, as long as we do not force extremely rare events to occur (in this case an outcome for Y very far from the expectation $E[Y]$), we can virtually ignore the effect of floating point precision and use the built-in machine floating point arithmetic. As a remark, results from simulations must be regarded with high scepticism anyway. Both, this algorithm and the generic algorithm 3.2.1 depend on the quality of the random number generator.

Preprocessing

We recursively split the sequence of random variables X_i in such a way that its partitions form a balanced binary tree: For $0 \leq i \leq j < n$ we define $Y_{ij} := X_i + X_{i+1} + \dots + X_j$. Thus we have $Y = Y_{0,n-1}$ and $X_i = Y_{ii}$ for $0 \leq i < n$. Also, for $0 \leq i \leq k \leq j < n$ we have

$$\text{Prob}(Y_{ij} = c) = \sum_{a+b=c} \text{Prob}(Y_{ik} = a) \text{Prob}(Y_{kj} = b) . \quad (3.1)$$

Now let T be the smallest set with the following properties: $Y = Y_{0,n-1} \in T$ and for each $Y_{ij} \in T$ with $i \neq j$ there are $Y_{i, \lfloor (i+j)/2 \rfloor} \in T$ and $Y_{\lfloor (i+j)/2 \rfloor + 1, j} \in T$. As a consequence, all the $X_i = Y_{ii}$ are elements of T . Next, for all $Y_{ij} \in T$ we compute their distribution

$$D(Y_{ij}) := (\text{Prob}(Y_{ij} = 0), \text{Prob}(Y_{ij} = 1), \dots, \text{Prob}(Y_{ij} = j - i + 1)) .$$

Since 3.1 shows that $D(Y_{ij})$ is the convolution of $D(Y_{i, \lfloor (i+j)/2 \rfloor})$ and $D(Y_{\lfloor (i+j)/2 \rfloor + 1, j})$, we can compute all the distributions in $O(n(\log n)^2)$ time by using FFT for the convolution products. We store the distributions in a binary tree such that each $D(Y_{ij})$ is in the parent node of the nodes for $D(Y_{i, \lfloor (i+j)/2 \rfloor})$ and $D(Y_{\lfloor (i+j)/2 \rfloor + 1, j})$. This binary tree needs $O(n \log n)$ space. Finally, we integrate the distribution of $D(Y) = D(Y_{0,n-1})$, that is, instead of $D(Y)$ we store the sums

$$\sum_{k=0}^i (D(Y))_k = \sum_{k=0}^i \text{Prob}(Y = k)$$

for all $0 \leq i \leq n$. Note that the sum above is very sensitive to roundoff errors because some values are extremely small compared to others. The reason why we only integrate $D(Y)$ is that later we want to generate random numbers e with the distribution of Y . Other datastructures might serve the purpose better, especially if we can not ignore rare outcomes for Y . One solution would be to split $D(Y)$ recursively

and to store the partial sums in a binary tree. The sums are then stored relative to each other in such a way that each one can be computed by summation along the path from the root to its node. This allows that the precision — as much as the convolution procedure has left to us — at the tails of $D(Y)$ will not be lost and can be used for a refined generation of e .

Choosing the Items

As already described, at first our algorithm decides how many items it will choose, i.e. it generates a random number e with the distribution of Y . This can be achieved by generating a random number in the interval $[0, 1]$ with uniform distribution and then performing a binary search in the array that stores the sums $\sum_{k=0}^i (D(Y))_k$ (because this is only a small part of the algorithm, there is little point in replacing the binary search by a Newton iteration). Next we call the procedure $\text{CHOOSE}(i, j, e)$ for $i = 0$ and $j = n - 1$. The procedure $\text{CHOOSE}(i, j, e)$ picks exactly e items from the set $\{i, i + 1, i + 2, \dots, j\}$ according to the probabilities for $X_k = 1$ under the condition that $X_i + \dots + X_j = e$. $\text{CHOOSE}(i, j, e)$ works as follows: If $e = 0$ we pick no item at all. Otherwise we check if $i = j$ and if yes, we have no other choice but to pick the only possible item i (in this case e must be 1, if not something went wrong with the implementation of the algorithm). In all other cases, we set $m = \lfloor (i + j)/2 \rfloor$ and compute the conditioned probabilities $\text{Prob}(Y_{im} = k | Y_{ij} = e)$ for $k = 0, 1, \dots, e$. We use Bayes' rule:

$$\text{Prob}(Y_{im} = k | Y_{ij} = e) = \frac{\text{Prob}(Y_{im} = k \wedge Y_{ij} = e)}{\text{Prob}(Y_{ij} = e)} = \frac{\text{Prob}(Y_{im} = k) \cdot \text{Prob}(Y_{m+1,j} = e - k)}{\text{Prob}(Y_{ij} = e)}.$$

The right part of the equation follows from the independence of our Poisson trials and the fact that $Y_{ij} = Y_{im} + Y_{m+1,j}$. Then we generate a random number f with the distribution of $Y_{im} | Y_{ij} = e$: We integrate the distribution (ie the the probabilities $\text{Prob}(Y_{im} = k | Y_{ij} = e)$), generate a random number in $[0, 1]$ uniformly, and locate it in the integral of the distribution. The values we finally pick are the values picked by the two subsequent calls $\text{CHOOSE}(i, m, f)$ and $\text{CHOOSE}(m + 1, j, e - f)$.

Sometimes we might wish to replace $\text{RANDOM}[0, 1]$ by a random number generator that generates (more or less perfect) random bits. In this case, we start with the interval $[0, 1]$ and choose (depending on the outcome of the first random bit) its lower or upper half. We repeat this procedure recursively until our interval is completely covered by two successive values of the integral of the distribution we want to sample from. Most of the time, this procedure terminates quickly. In some very rare cases (with nearly zero probability), it can run forever but we can break up if the error is low enough.

Performance

We assume constant time for all arithmetic operations and negligible time for $\text{RANDOM}[0, 1]$. For $N > 1$ and arbitrary i , the procedure $\text{CHOOSE}(i, i + N, e)$ takes time $T(N, e) \leq ce + T(\lfloor N/2 \rfloor, f) + T(N - \lfloor N/2 \rfloor, e - f)$, depending on the outcome of the randomly generated f . The constant c must be large enough so that it bounds $T(0, e)$ while at the same time ce bounds the time needed for computing the con-

ditioned probabilities and for generating f . Substituting the times $T(\lfloor N/2 \rfloor, f)$ and $T(N - \lfloor N/2 \rfloor, e - f)$ and also the times for all the other recursive calls of CHOOSE(...) yields

$$\begin{aligned} T(N, e) &\leq ce + cf + c(e - f) + cf_1 + c(f - f_1) + cf_2 + c(e - f - f_2) + \dots \\ &\quad + ce \\ &= ce + ce + ce + \dots + ce. \end{aligned}$$

Here, f_1 and f_2 are the outcomes for f in the first two subsequent calls. However, the maximum time spent at the level of the first two subsequent calls sums up to $cf + c(e - f) = ce$ as it also happens at each other level of our recursion, no matter what the respective outcomes for f are. Altogether we have $O(\log N)$ times the term ce , hence $T(N, e) = O(e \log N)$. We conclude that the expected time for our algorithm is $O(E[Y] \log n)$. Also, we need no more than $e \lceil \log n \rceil + 1$ calls of RANDOM[0,1]: There are e items to pick, which in the worst case have to be located separately by using $\lceil \log n \rceil$ random numbers for each. Hence the expected number of calls of RANDOM[0,1] is $E[Y] \lceil \log n \rceil + 1$. As previously mentioned, the algorithm makes use of a precomputed datastructure which takes $O(n \log n)$ space and $O(n(\log n)^2)$ time for construction.

However, great savings in space and particularly in time needed for the preprocessing can be made by observing that many entries in the joint distributions are practically zero. The maximum number of interest for the positive outcomes is typically much smaller than n .

Results and Suggestions

We implemented both, the generic algorithm 3.2.1 and the algorithm described here in the JAVA programming language. We used the built-in pseudorandom number generator which generates a floating point number in the intervall $[0, 1]$ with uniform distribution. All arithmetic was done by the machine's floating point unit (DOUBLES) with no special regard to roundoff errors. Then we run both algorithms for $n = 10000$ on a 800 MHz pentium processor: The generic algorithm 3.2.1 used 3.6 ms independently of $E[Y]$ while the preprocessing for our algorithm used 215 ms (using a simple radix-2 FFT for the convolution products). For $E[Y]/n = 0.0002/0.002/0.02/0.2/0.5$, time consumption of our algorithm averaged 0.033/0.18/1.07/5.59/7.31 ms respectively. This confirms our expectation that our algorithm outperforms the generic algorithm 3.2.1 if $E[Y]$ is small compared to n and the number of repetitions is large. But even for $E[Y]/n = 0.5$, choosing from $n = 10000$ items only takes twice as much time as with the generic algorithm. Since we can always choose all $p_i \leq 0.5$ and since the logarithmic factor for other practical values for $n > 10000$ will not make much difference, our algorithm appears to be comparable to the generic algorithm in all cases with high speedup for small $E[Y]$. Plus, it has the advantage that it is possible to fix the number of positive outcomes in each sample in advance.

There are, of course, other algorithms for simulating many parallel Bernoulli trials. For instance, we could for each X_i precompute a list of future outcomes and put them into a queue. This algorithm would have the advantage that it adapts naturally to updates on the set of all trials without the time-consuming FFT. So this algorithm will be asymptotically much faster than ours. For interactive simu-

lations, for instance in computer games, preparing the queue for future outcomes for each trial X_i can result in a constant-factor overhead which leads to unpleasant delays when all X_i are initialized. In this case we suggest to use the alg+orithm from this subsection in order to get a quick start for the initial simulation and then preparing the queues for the adaptive algorithm in the background.

3.2.2 Interactive Line Breaking

Our next application is interactive line breaking in text processors. One way to break paragraphs into lines is to scan the text (or part of it) from left to right and add a line break at a position with the best penalty for the current line. This does not take into account that a good choice for the current line might lead to a bad overall line breaking for subsequent lines, thus even leading to a suboptimal overall solution for the whole paragraph. A very good line breaking algorithm is Knuth's best total-fit line breaking which works offline by constructing the best total-fit while scanning the paragraph from its start. However, for long paragraphs this might not always be feasible for interactive text processors, which is especially true if more complicated tasks such as score breaking, page breaking, or image layout have to be included.

We give here a simple version of an interactive line breaking algorithm for best total-fit solutions in the very basic setting of inserting line breaks into paragraphs.

Let X be a finite alphabet in which the (unbroken) paragraph has been written. An additional symbol $\$ \notin X$ shall be understood as a line break. Given a paragraph $w \in X^*$ the task is to find a string $w' = w_0\$w_1\$ \dots \$w_{n-1}\$w_n \in (X \cup \$)^*$, $w_i \in X^*$, which breaks $w = w_0 \dots w_n$ into lines such that w' has a minimum value for its line breaking penalty $\sigma(w') \geq 0$. This minimum penalty of w is denoted by $\sigma_{\min}(w)$. We have to make a few assumptions about the penalty function σ : First we assume that for an unbroken line $u \in X^*$ we get a penalty $\sigma(u) \geq 0$. In this case, $\sigma(u)$ evaluates the appearance of the string u layouted on a single line. Secondly, if x is of the form $x = u\$v$ for $u, v \in (X \cup \$)^*$, we assume that the penalty function is monotone:

$$\sigma(v') \geq \sigma(v) \wedge \sigma(u') \geq \sigma(u) \quad \Rightarrow \quad \sigma(v'\$u') \geq \sigma(v\$u)$$

The assumption is reasonable since it means that a line breaking cannot be made better by replacing one of its parts by a worse broken part while leaving all other line breaks as they are.

In most texts, the maximum length of a single line is bounded by some number k . This means that for any given $w = w_1 \dots w_n$, $w_i \in X$ with $n > 2k$, there must be a line break within the first k and within the last k symbols of each feasible line breaking of w . For $1 \leq i, j \leq k$, we set

$$\delta(w, i, j) := \sigma_{\min}(w_{i+1} \dots w_{n-j}) .$$

Let $u = u_1 \dots u_m, v = v_1 \dots v_n, u_i, v_i \in X$ be two unbroken paragraphs of length $> 2k$. Let c_{ij} be some $k \times k$ matrix over $(X \cup \$)^*$ with $\sigma(c_{ij}) = \delta(u, i, j)$ and such that c_{ij} is a line breaking of the infix $u_i \dots u_{m-j}$ of u . Let d_{ij} be a similar matrix for v . Then we have

$$\delta(uv, i, j) = \min_{r,s \in 1..k} \sigma(c_{ir}\$u_{m-r+1} \dots u_m v_1 \dots v_{s-1}\$d_{sj}) . \quad (3.2)$$

Usually, when $\sigma(c_{ir})$ and $\sigma(d_{sj})$ is known, computing $\sigma(c_{ir} \$ u_{m-r+1} \cdots u_m v_1 \cdots v_{s-1} \$ d_{sj})$ needs only to take the line $u_m v_1 \cdots v_{s-1}$ into account. We assume $O(k)$ time for that, but sometimes costs can be higher.

Let w be an unbroken paragraph of length n which is given as a sequence of a persistent array's A elements. As usual, A is represented as a binary tree and thus permits edit operations such as insertion, deletion, and concatenation of paragraphs. In order to maintain a line breaking with minimum penalty, we memorize for each inner node p of the tree representation the following information: Let $u = u_1 \cdots u_m$, $u_i \in X$ be the infix of w which corresponds to the subtree at node p . At p , we store all values $\delta(u, i, j)$, $i, j = 1..k$ and corresponding optimal line breakings c_{ij} of the infix $u_i \cdots u_{m-j}$. According to equation 3.2, all those values for p depend solely on the values for its children. The computation of each of its k^2 entries needs to test k^2 line breaks. Each such test involves the time $O(k)$ for the evaluation of the line breaking, and $O(\log n)$ time for concatenating the newly broken paragraphs. Altogether, each update on w needs $O(k^4(k + \log n))$ time for updating to a new optimal line breaking.

The whole procedure can be cast into the homomorphism-framework by identifying line breakings of the same original paragraph which have in the same penalty. The factor k^5 might appear prohibitive. But first note that not all positions in a line are candidates for a line break, so k is often very small. Secondly, by making the boundaries overlap, we can get rid of one of the two middle line breaks in equation 3.2, which saves a factor k .

3.2.3 Related Work

Decomposing arrays into trees and using monoid homomorphisms is an old but very versatile technique. The same goes for the fingerprint technique modulo random primes. Historical notes for the latter can be found in [MoRa 95], but we want to add that the fingerprint technique has also been adapted for testing equality of certain binary decision diagrams. The use for persistent arrays, however, is new and has been first described in [Bar 04].

A best-total-fit line breaking algorithm with all bells and whistles has been described in [KnPl 81]. A similar contribution is the thesis [Ren 02] for the case of musical scores. Both algorithms are noninteractive, but the ideas presented here are rooted in discussions with the author K. Renz. Almost all algorithms (except line breaking) from this chapter have been implemented either by N. Barraci or by the author, see [Bar 04]. In the latter work, generalized state machines have been employed for space-efficient computation with L -systems.

Chapter 4

Language-Restrictions and Minimal Syntax-Trees

The well known travelling-salesman-problem is to find a shortest closed path which visits each node of a given graph. This problem is NP-hard. If the order of the traversal of the nodes is given, the problem reduces to finding a shortest path between each pair of consecutive nodes and becomes polynomial-time solvable. On the other hand, we might assume further restrictions which in turn make the problem more difficult: A travelling saleswoman (see [Gon 95]) might have a certain schedule, such that the order of the visits is fixed. However, she might be able to choose between different fares of different transportation companies. The companies might bundle certain tickets and offer special prices for certain routes. Those problems — and also the maze from the introduction in the first chapter — can often be modelled by context-free language restrictions on the set of feasible paths.

Informally stated, for a language $L \subseteq X^*$ a L -restricted path problem is a path problem which edges are labelled with symbols from X . We then consider only those paths as valid where the consecutive edge-labels form a string in L . Solutions to context-free restricted path problems will also give us an example for polynomial-time constructable, exponentially long paths. They are a prime example for exponentially long arrays which are representable in polynomial space by our persistent array implementation.¹

In this chapter we show how — under certain conditions — the context-free language-restricted-path problem can be solved in polynomial time. Again, we will use the semiring framework introduced in section 2.2. This gives us an abstraction for many combinatorial path problems, for instance counting the number of possible paths (for unambiguous context-free grammars), construction of solutions, etc. Furthermore, our general approach can help finding pseudo-polynomial time algorithms for hard problems. We will give examples later on, when discussing the knapsack problem and the delay-bounded shortest-path problem.

¹Another example are strings generated by L0-grammars in fractal image generation, which is not considered in this thesis, see instead [Bar 04].

4.1 Semiring-Valued Context-Free Grammars

We denote a context-free grammar P by a tuple $P = (X, V, P, S)$ where X is a finite set of terminal letters, V is the finite set of nonterminals, S is the initial variable, and $P \subseteq V \times (X \cup V)^*$ is a finite set of productions. We also demand $V \cap X = \emptyset$. P is in Chomsky normalform if $P \subseteq V \times (X \cup VV)$. Likewise, a **semiring-valued context-free grammar** P for a semiring $R = (R, +, \cdot, 0, 1)$ is a tuple $P = (X, V, p, S)$ where X, V and S have the same meaning as before and p is a function from $P \subseteq V \times (X \cup V)^*$ to R which is 0 in all but finitely many cases. P is in Chomsky normalform if

$$p(A \rightarrow v) \neq 0 \quad \Rightarrow \quad v \in X \cup VV$$

holds for $A \in V, v \in (X \cup V)^*$.

In order to distinguish context-free grammars from R -valued context-free grammars, we often call the former a *plain* context-free grammar. This convention is also used for parsers, languages, etc. A **R -valued language** over the alphabet X is a map $L : X^* \rightarrow R$. For $x \in X^*$, we define $L_x^P = L_x$ as the language which maps x to 1 and all other strings to 0. For two R -valued languages L_1, L_2 over X their **Cauchy product** $L_1 \cdot L_2$ is defined as

$$(L_1 \cdot L_2)(w) := \sum_{uv=w} L_1(u) \cdot L_2(v) \quad (w \in X^*).$$

In case of the boolean semiring, the Cauchy product corresponds to the concatenation of two languages. Let $P = (X, V, p, S)$ be a R -valued context-free grammar. Provided that R has sufficient closure properties, for each nonterminal $A \in V$ a R -valued language L_A can be defined: We consider the system of equations

$$L_A^P = \sum_{n \geq 0} \sum_{x_1, \dots, x_n \in (X \cup V)} p(A \rightarrow x_1 \cdots x_n) \cdot (L_{x_1}^P \cdots L_{x_n}^P) \quad (A \in V) \quad (4.1)$$

such that the languages $L_A^P, A \in V$ can be defined as the smallest solution-vector (in the sense of the least number of terms in the formal sums) of 4.1. However, instead of following this algebraic approach, we define $L_A^P, A \in V$ in terms of weighted syntax-trees, which is appropriate for R -valued *context-free* grammars:

Definition 8 *Given a set X of terminals and a set V of nonterminals, the set of all possible **syntax-trees** over X, V is defined as the smallest set $\mathcal{T}(X, V)$ with:*

- (i) $X \cup \{\varepsilon\} \subseteq \mathcal{T}(X, V)$.

We set $\text{root}(x) := \text{front}(x) := x$ for all $x \in X \cup \{\varepsilon\}$.

- (ii) $\forall k > 0, A \in V, T_1, \dots, T_k \in \mathcal{T}(X, V). \quad (A, T_1, \dots, T_k) \in \mathcal{T}(X, V)$.

We then set $\text{root}(A, T_1, \dots, T_k) := A$ and $\text{front}(A, T_1, \dots, T_k) := \text{front}(T_1) \dots \text{front}(T_k)$

Informally, for a syntax-tree $T \in \mathcal{T}(X, V)$, $\text{root}(T)$ denotes its topmost nonterminal and $\text{front}(T)$ denotes its concatenation of its terminal letters (leaves) in their natural order. The weight function p of

a given R -valued context-free grammar $P = (X, V, p, S)$ can be extended to syntax-trees over X, V by setting

$$p(T) := 1$$

for syntax-trees $T \in X \cup \{\varepsilon\}$ and

$$p(T) := p(A \rightarrow \text{root}(T_1) \cdots \text{root}(T_k)) \cdot p(T_1) \cdots p(T_k)$$

for syntax-trees $T = (A, T_1, \dots, T_k)$ with $A \in V$, $T_1, \dots, T_k \in \mathcal{T}(G)$, $k > 0$. Thus the weight of a syntax-tree T is the weight of its topmost production, multiplied by the weights of the root's subtrees from left to right. This is the same as multiplying the productions of T 's corresponding left-most derivation from left to right. Finally, we set for $w \in X^*$ and $A \in V$:

$$L_A^P(w) := \sum_{\substack{\text{root}(T)=A \\ \text{front}(T)=w}} p(T).$$

The languages L_A^P defined in this way exist for all ω -complete semirings R . Alternatively L_A^P exists if in P all ε -productions have value zero. One can verify that they satisfy the equational system 4.1. The language which is generated by the grammar P is defined as $L(P) := L_S^P$, i.e. the language which is generated by P 's initial nonterminal symbol S .

From now on, we will only consider R -valued grammars which are already in Chomsky normalform. In this case we have by 4.1

$$L_A^P = \sum_{x \in X} p(A \rightarrow x) \cdot L_x^P + \sum_{B, C \in V} p(A \rightarrow BC) \cdot L_B^P \cdot L_C^P,$$

where $L_B^P \cdot L_C^P$ is the Cauchy-product of the two R -valued languages. It follows immediatly

$$L_A^P(x) = p(A \rightarrow x)$$

for $x \in X$ and

$$L_A^P(w) = \sum_{B, C \in V} \sum_{uv=w} p(A \rightarrow BC) \cdot L_B^P(u) \cdot L_C^P(v)$$

for $w \in XX^+$.

For the boolean semiring $\mathbb{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$ there is a natural correspondence between a \mathbb{B} -valued context-free grammar and a plain context-free grammar by removing all zero-valued productions. In this case, a transformation into Chomsky normalform needs only polynomial time. Note that in the general case, a weight-preserving transformation of a context-free grammar P into a grammar in Chomsky normalform P' with $L(P)(w) = L_{P'}(w)$ for all $w \in X^+$ depends on the properties — like convergence properties — of the underlying semiring. In case of a non-commutative semiring and the occurrence of ε -productions, a Chomsky normalform might not even exist. However, for most interesting semirings a Chomsky normalform can be efficiently computed for any context-free grammar. For more details, see Goodman's thesis [Goo 98].

4.2 Language-Restrictions and Minimum Syntax-Tree

Definition 9 For a finite alphabet X , the context-free language-restricted all-pairs-shortest-path (CF-APSP) problem has as its input a connected directed graph $G = (N, E)$ with edges labelled by $\delta : E \rightarrow \mathbb{R}^+$ and by $\phi : E \rightarrow X$. Further, a language $L(P) \subseteq X^*$ is given by a plain context-free grammar $P = (X, V, P, S)$. The task is to find, for each pair of nodes $s, t \in N$, a path $\pi = d(s, t)$ with $\phi(\pi) \in L(P)$ such that $\delta(\pi)$ is minimal.

Note that, similar to section 2.2.1, paths are represented by sequences of edges and that edge-labels (with values in a monoid) are extended to paths naturally by multiplying (or adding) the values of the traversed edges.

Like the APSP problem in section 2.2.1, the CF-APSP problem can also be generalized to an algebraic context-free restricted all-pairs-shortest-distance (CF-APSD) problem for semiring-valued graphs:

Definition 10 Let X be a finite alphabet and $R = (R, +, \cdot, 0, 1)$ be some semiring. The R -valued context-free language-restricted all-pairs-shortest-distance (CF-APSD) problem has as its input a complete graph $G = (N, E)$ — that is, $E = N \times N$ — with edges labelled by $\delta : E \rightarrow R$ and by $\phi : E \rightarrow X$. Further, a language $L(P) \subseteq X^*$ is given by a plain context-free grammar $P = (X, V, P, S)$. The task is to compute, for each pair of nodes $s, t \in N$, a minimum distance $d(s, t)$ of all paths π from s to t such that $\phi(\pi) \in L(P)$. More formally:

$$d(s, t) := \sum_{\substack{\pi: s \rightarrow t \\ \phi(\pi) \in L(P)}} \delta(\pi)$$

Choosing $R = (E^* \cup \infty, \min, \cdot, \infty, \epsilon)$ to be the semiring for concatenating paths which was defined in section 2.2, one can see that the CF-APSP problem is an instance of the semiring-valued CF-APSD problem. For most practical purposes, the tie-breaking lexicographic comparison of the min-operation is not necessary. However, since then the resulting structure is not a semiring anymore, correctness of the resulting algorithms must be proven separately. On the other hand, the probabilistic algorithms from section 3.1.1 provide means in order to determine the lexicographically first one among the minimum paths with sufficient efficiency and reliability.

As an interesting property of the CF-APSP problem the resulting paths can be exponentially long, as illustrated by the following example:

Example 1 Let $P = (\{a\}, \{A_0, \dots, A_n\}, P, A_0)$ be the context-free grammar with the productions

$$P := \{A_{i-1} \rightarrow A_i A_i : i = 1..k\} \cup \{A_0 \rightarrow a\}.$$

The graph $G = (N, E)$ consists of a single node $N = \{1\}$ and the edge $(1, 1)$ is labelled by the letter $\phi((1, 1)) = a$ while its cost is $\delta((1, 1)) = 1$. Now, for each n , the shortest path π from 1 to itself with $\phi(\pi) \in L(P)$ is the one with traverses the edge $(1, 1)$ exactly 2^{n+1} many times, thus producing the (only) string $a^{(2^{n+1})} \in L(P)$.

Although the example above is trivial, it clearly shows that exponentially long paths have to be expected. It also raises the question under which circumstances what kind of growth might occur. The question is not within the scope of this thesis and may be left for further research.

We now turn to a related problem, where a path's cost is not given by the costs of its edges but by its corresponding syntax-trees.

Definition 11 *Let X be a finite alphabet and $R = (R, +, \cdot, 0, 1)$ be some semiring. The R -valued all-pairs-minimum-syntax-tree (APMS) problem has as its input a complete graph $G = (N, E)$ with edges labelled by $\phi : E \rightarrow X$. Further a R -valued context-free grammar $P = (X, V, p, S)$ is given in Chomsky normalform. The task is to compute, for each nonterminal $A \in V$ and each pair of nodes $s, t \in N$, the minimum weight $c(A, s, t)$ of all syntax-trees T with $\text{root}(T) = A$ and $\text{front}(T) = \phi(\pi)$ for some path $\pi : s \rightarrow t$:*

$$c(A, s, t) := \sum_{\pi: s \rightarrow t} L(P)(\phi(\pi))$$

4.2.1 Reducing the CF-APSD Problem to the APMS Problem

The context-free restricted APSD problem can, for certain semirings or languages, be reduced to the all-pair-minimum-syntax-tree problem. This is true in the case of

1. unambiguous context-free grammars, or alternatively
2. idempotent semirings, i.e. $a + a = a$ holds for all $a \in R$.

Proposition 18 *Let $R = (R, +, \cdot, 0, 1)$ be a semiring and $P = (X, V, P, S)$ be a (plain) context-free grammar in Chomsky normalform. Assume P being unambiguous or R being idempotent. Then the CF-APSD problem reduces to the context-free APMS Problem within a polynomial number of operations.*

PROOF: Let R and P be as stated in the proposition and (G, E) be a graph with edge labels $\delta : E \rightarrow R$ and $\phi : E \rightarrow X$. Together they form an input instance of the CF-APSD problem. Let $d(s, t)$, $s, t \in N$ be the solution of the corresponding CF-APSD problem with input P, G , and δ, ϕ . First we note that we can assume P being Chomsky normalform. It is always possible to transform an unambiguous context-free grammar into Chomsky normalform by retaining unambiguity.

We then transform the (plain) context-free grammar $P = (X, V, P, S)$ into a R -valued context-free grammar $P' = (X', V, p', S)$ with $X' = E$. We set

$$p'(A \rightarrow (s, t)) := \begin{cases} \delta((s, t)), & \text{if } A \rightarrow \phi((s, t)) \in P \\ 0, & \text{else} \end{cases}$$

for all edges $(s, t) \in E$ and all nonterminals $A \in V$. Further, we set

$$p'(A \rightarrow BC) := \begin{cases} 1, & \text{if } A \rightarrow BC \in P \\ 0, & \text{else} \end{cases}$$

for all productions $A \rightarrow BC$ from P . Let T be a syntax-tree over $X' = E$ and V . Let T' be the syntax-tree over X, V resulting from T by replacing all its leaves $(s, t) \in X' = E$ by their corresponding letter $\phi((s, t))$. From the definition above it follows

$$p'(T) = \begin{cases} \delta(\text{front}(T)), & \text{if } T' \text{ is a valid syntax-tree for the original grammar } P \\ 0, & \text{else} \end{cases} \quad (4.2)$$

As the input for the APMS problem we chose P', G , together with the identity map $(s, t) \mapsto (s, t)$ as the edge labelling. Let $c(A, s, t)$, $A \in V$, $s, t \in N$ be the corresponding solution. Then we have for all $s, t \in N$:

$$\begin{aligned} c(S, s, t) &= \sum_{\pi: s \rightarrow t} L(P')(\pi) \\ &= \sum_{\pi: s \rightarrow t} \sum_{\substack{T \in \mathcal{T}(E, V) \\ \text{root}(T) = S \wedge \text{front}(T) = \pi}} p'(T) \\ &= \sum_{\pi: s \rightarrow t} \sum_{\substack{T \in \mathcal{T}(E, V) \wedge T' \text{ is valid in } P \\ \text{root}(T) = S \wedge \text{front}(T) = \pi}} \delta(\pi) \quad (\text{by equation 4.2}) \end{aligned}$$

Assume that for the path $\pi: s \rightarrow t$ it is $\phi(\pi) \notin L(P)$. In this case no syntax tree $T \in \mathcal{T}(E, V)$ with root S exists such that replacing its leaves $(s, t) \in E$ by their corresponding values $\phi((s, t))$ yields a valid syntax-tree in P . We get

$$\sum_{\substack{T \in \mathcal{T}(E, V) \wedge T' \text{ is valid in } P \\ \text{root}(T) = S \wedge \text{front}(T) = \pi}} \delta(\pi) = 0.$$

On the other hand let $\phi(\pi) \in L(P)$. Assume the original grammar P is unambiguous. We want to show that there are no syntax-trees $T_1 \neq T_2$ over E, V , both with root S and front $\phi(\pi)$, such that their corresponding T'_1, T'_2 are valid in P . Since T_1 and T_2 have the same front, from $T_1 \neq T_2$ it follows that replacing their leaves by the map $(s, t) \mapsto \phi((s, t))$ cannot make the resulting syntax-trees T'_1, T'_2 equal — like T_1, T_2 they must differ somewhere in their inner structure. Hence T'_1, T'_2 are two different valid syntax-trees in P for $\phi(\pi)$, which contradicts that P is unambiguous. In other words there must be exactly one syntax-tree T over E, V with root S and front ϕ , hence

$$\sum_{\substack{T \in \mathcal{T}(E, V) \wedge T' \text{ is valid in } P \\ \text{root}(T) = S \wedge \text{front}(T) = \pi}} \delta(\pi) = \delta(\pi).$$

If in the other case $\phi(\pi) \in L(P)$ and R is idempotent, then the actual number of syntax-trees does not matter if there is at least one. Again the sum evaluates to $\delta(\pi)$. Putting it all together we find

$$\sum_{\pi: s \rightarrow t} \sum_{\substack{T \in \mathcal{T}(E, V) \wedge T' \text{ is valid in } P \\ \text{root}(T) = S \wedge \text{front}(T) = \pi}} \delta(\pi) = \sum_{\substack{\pi: s \rightarrow t \\ \phi(\pi) \in L(P)}} \delta(\pi) = d(s, t).$$

From this we conclude $d(s, t) = c(S, s, t)$ for all $s, t \in N$. So we obtain the solutions for the context-free restricted all-pairs-shortest-distance problem instances from the solution to the corresponding all-pairs-minimum-syntax-tree problem instance. \square

4.3 Solving The All-Pairs-Minimum-Syntax-Tree Problem

The definition of the values $c(A, s, t)$ in definition 11 gives rise to the following recursion for $A \in V$ and $s, t \in N$:

$$c(A, s, t) = p(A \rightarrow \phi((s, t))) + \sum_{r \in N} \sum_{A \rightarrow BC} c(B, s, r) \cdot c(C, r, t) \quad (4.3)$$

In order to see this, we first write

$$c(A, s, t) = p(A \rightarrow \phi((s, t))) + \sum_{\substack{\pi: s \rightarrow t \\ |\pi| \geq 2}} L_A^P(\phi(\pi)) .$$

By employing that the grammar P is in Chomsky normalform we conclude for the second term in the sum above:

$$\begin{aligned} \sum_{\substack{\pi: s \rightarrow t \\ |\pi| \geq 2}} L_A^P(\phi(\pi)) &= \sum_{\substack{\pi: s \rightarrow t \\ |\pi| \geq 2}} \sum_{uv = \phi(\pi)} \sum_{A \rightarrow BC} L_B^P(u) \cdot L_C^P(v) \\ &= \sum_{\pi: s \rightarrow t} \sum_{\pi' \cdot \pi'' = \pi} \sum_{A \rightarrow BC} L_B^P(\phi(\pi')) \cdot L_C^P(\phi(\pi'')) \\ &= \sum_{r \in N} \sum_{A \rightarrow BC} \sum_{\pi': s \rightarrow r} \sum_{\pi'': r \rightarrow t} L_B^P(\phi(\pi')) \cdot L_C^P(\phi(\pi'')) \\ &= \sum_{r \in N} \sum_{A \rightarrow BC} \left(\sum_{\pi': s \rightarrow r} L_B^P(\phi(\pi')) \right) \cdot \left(\sum_{\pi'': r \rightarrow t} L_C^P(\phi(\pi'')) \right) \\ &= \sum_{r \in N} \sum_{A \rightarrow BC} c(B, s, r) \cdot c(C, r, t) \end{aligned}$$

Note that for zero-length paths π we always have $L_A^P(\phi(\pi)) = 0$ since P is in Chomsky normalform, hence they neither got lost nor introduced any extra-cost during the calculation above.

The recursion 4.3 is employed in algorithm 4.1. For many semirings, the algorithm is only a partial algorithm, since it lacks of a proper termination condition and sometimes may never terminate. We will, however, establish termination after polynomial time for two important classes of problem instances: The algorithm always terminates for finite context-free languages after a polynomial number of operations, not depending on the semiring. Other important examples are the tropical semirings. We generalize this by the following definition:

Definition 12 A semiring $R = (R, \min, +, \infty, 0)$ is **bounded-tropical** if there exists a linear order \leq on R such that for $a, b \in R$, $\min(a, b)$ returns their minimum and also $\min(a, a + b) = a$.

Important examples for bounded-tropical semirings are:

- (i) The tropical semiring $(\mathbb{R}^+ \cup \infty, \min, +, \infty, 0)$.
- (ii) The semiring $([0..1], \max, \cdot, 0, 1)$, where $[0..1]$ denotes the unit interval.

As a counter-example, the semiring $(\mathbb{R} \cup \infty, \min, +, \infty, 0)$ is idempotent but not bounded-tropical. It cannot be made bounded-tropical by any linear order. One can see this by constructing a graph with a negative loop such that algorithm 4.1 does not terminate, which contradicts our subsequent results.

Definition 13 For a semiring $R = (R, +, \cdot, 0, 1)$, a R -valued language $L : X^* \rightarrow R$ is a **finite language** if $L(w) \neq 0$ for only finitely many $w \in X^*$.

At a first glance, finite context-free languages appear uninteresting since the underlying language is automatically a regular language. However, note that the complexity of an algorithm usually depends on the size of the grammar. If we measure the length of a grammar by the number of symbols of all productions together, then a regular grammar for a finite language must be at least as long as the longest string which it generates. This is not generally true for many context-free grammars. Indeed, we give some interesting examples in section 4.3.1, which have applications in some combinatorial problems on graphs.

Algorithm 4.1 Algorithm for the APMS problem, see definition 11

```

 $k := 0$ 
for  $(A \in V, s, t \in N)$  do
     $c^{(0)}(A, s, t) := 0$ 
end for
 $stop := false$ 
while not  $stop$  do
     $k := k + 1$ 
    for  $(A \in V, s, t \in N)$  do
         $c^{(k)}(A, s, t) := p(A \rightarrow (s, t))$ 
        for  $(B, C \in V, r \in N)$  do
             $c^{(k)}(A, s, t) := c^{(k)}(A, s, t) + c^{(k-1)}(B, s, r) \cdot c^{(k-1)}(C, r, t)$ 
        end for
    end for
    determine whether values are stable:
     $stop := true$ 
    for  $(A \in V, s, t \in N)$  do
        if  $c^{(k)}(A, s, t) \neq c^{(k-1)}(A, s, t)$  then
             $stop := false$ 
        end if
    end for
end while
    
```

Algorithm 4.1 can compute meaningful values even in those cases where its termination condition is never satisfied. For instance, for the semiring $([0, 1], +, \cdot, 0, 1)$ of the unit interval of real numbers,

convergence of the sequence $(c^{(k)})_{k \in \mathbb{N}}$ can be sufficiently fast if the initial values are bounded away from 1. One can even guarantee a polynomial number of operations in the two following cases:

Proposition 19 *Let R be a semiring. Given a semiring $R = (R, +, \cdot, 0, 1)$, a R -valued context-free grammar in Chomsky normalform $P = (V, X, p, S)$ and a graph $G = (N, E)$ with edge-labelling $\phi : E \rightarrow 2^X$. Then the algorithm 4.1 for the APMS problem always terminates if*

- (i) *R is bounded-tropical. In this case the number of operations is $O(|V|^4|N|^5)$ (but a slight modifications leads to smaller exponents), or*
- (ii) *$L(P)$ is finite. In this case the number of operations is $O(|V|^3|N|^3d_{\max})$. Here d_{\max} is the maximum depth of a non-zero syntax-tree, which means a syntax-tree $T \in \mathcal{T}(X, V)$ with $p(T) \neq 0$.*

Note that for all semirings considered in this thesis, we have $d_{\max} = O(|V|)$ since in all these cases a repetition of a variable along a path of a non-zero syntax-tree can be used in order to produce arbitrary long strings w with $L(w) \neq 0$.

Proof of proposition 19.

Case R is bounded-tropical: We set $I := \{(A, s, t) : A \in V, s, t \in N\}$. The bound follows from the fact that in each iteration k of the outer while-loop, at least one additional c -value attains its final value, that is there exists $A, s, t \in I$ such that $c^{(k)}(A, s, t) = c^{(k+m)}(A, s, t)$ for all $m \leq 0$. Since there are only $|V||N|^2$ many elements in I , and since the inner for-loops need $O(|V|^3|N|^3)$ many operations, the bound $O(|V|^4|N|^5)$ follows.

In order to show that in each iteration a c -value attains its final value, we set $M_0 := \emptyset$. In the k th iteration, $1 \leq k \leq |I|$, we choose an arbitrary triple $(A, s, t) \in I \setminus M_{k-1}$ such that $c^{(k)}(A, s, t) \leq c^{(k)}(A', s', t')$ for all other $(A', s', t') \in I \setminus M_{k-1}$. Then we set $M_k := M_{k-1} \cup (A, s, t)$. We show by induction that for all $(A, s, t) \in M_k$ $c^{(k)}(A, s, t)$ has attained the final value $c(A, s, t)$ by induction. For $k = 0$ this is trivial since M_0 is empty. In the k th iteration, $1 \leq k \leq |I|$, we only need to show our claim for the only new triple $(A, s, t) \in M_k \setminus M_{k-1}$, since for all others it is our inductive hypothesis. We assume for (A, s, t) the opposite, i.e.

$$c^{(k+m+1)}(A, s, t) < c^{(k+m)}(A, s, t)$$

for some $m \geq 0$. In this case it follows that there exists another triple (A', s', t') which's c -value has changed in the step immediately before:

$$c^{(k+m)}(A', s', t') < c^{(k+m-1)}(A', s', t') \text{ and } c^{(k+m)}(A', s', t') < c^{(k)}(A, s, t) .$$

By repeating this argument until $m = 0$, we find that there exist (A', s', t')

$$c^{(k)}(A', s', t') < c^{(k-1)}(A', s', t') \text{ and } c^{(k)}(A', s', t') < c^{(k)}(A, s, t) .$$

However, since A, s, t was chosen from $I \setminus M_{k-1}$ to have minimal value $c^{(k)}(A, s, t)$, it follows $(A', s', t') \in M_{k-1}$. But then $c^{(k)}(A', s', t') < c^{(k-1)}(A', s', t')$ contradicts the inductive assumption that

$c^{(k-1)}(A', s', t')$ has already attained the final value $c(A', s', t')$ for all $(A', s', t') \in M_{k-1}$.

The bound $O(|V|^4|N^5|)$ for the number of operations is not very attractive. We get a better algorithm by using Fibonacci heaps and also considering only relevant productions in the inner loops. The process is then similar to Dijkstra's single-source-shortest-path algorithm. See also the remarks in [BJM 98].

Case $L(P)$ is finite: We show that in the k th iteration of the outer while-loop, we have for $A \in V, s, t \in N$

$$c^{(k)}(A, s, t) = \sum_{\pi: s \rightarrow t} \sum_{\substack{\text{root}(T)=A \\ \text{front}(T)=\phi(\pi)}} p(T),$$

where T ranges over all syntax-trees of depth $\leq k$. This implies that $c^{(d_{\max})}(A, s, t)$ has the correct value of $c(A, s, t) = \sum_{\pi: s \rightarrow t} L(P)(\phi(\pi))$.

For $k = 1$, the contribution of smaller sub-trees is zero, since there are none. Hence equation 4.3 implies $c^{(k)}(A, s, t) = p(A \rightarrow \phi((s, t)))$, which is exactly what algorithm 4.1 computes. Else we assume that 4.4 already holds for k . One can then conclude that it also holds for $k + 1$ by using equation 4.3 again. For the time bound we get d_{\max} iterations of $O(|V|^3|N|^3)$ operations for the inner loops. \square

4.3.1 Finite Languages

In the context of the context-free restricted APSD problem and of the APMS problem, finite languages are of particular interest. This is especially true when we take into account the size of a grammar compared to the strings it generates.

For counting the occurrences of edges along a path which are marked with the letter $a \in X$, we define for $n, k \in \mathbb{N}$ the languages

$$\begin{aligned} L_{a,=n} &:= \{w \in X^+ : |w|_a = n\}, \\ L_{a,\leq n} &:= \{w \in X^+ : |w|_a \leq n\}, \\ L_{a,[n,n+k]} &:= \{w \in X^+ : n \leq |w|_a \leq n+k\}, \text{ and} \\ L_{a,\geq n} &:= \{w \in X^+ : |w|_a \geq n\}. \end{aligned}$$

Proposition 20 *Let be $a \in X$ and $n, k \in \mathbb{N}$. For $L = L_{a,=n}, L_{a,\leq n}, L_{a,[n,n+k]}, L_{a,\geq n}$ the following proposition holds:*

Each context-free grammar P with $L = L(P)$ is at least $\Omega((\log n)^{\frac{1}{k}})$ many productions for some $k \geq 1$. If P is in Chomsky normalform, P has at least $\log n$ many productions, and there exists such a grammar P with $O(\log n)$ many productions which is unambiguous.

PROOF: The languages $L = L_{a,=n}, L_{a,\leq n}, L_{a,[n,n+k]}$ all contain the word a^n and produce at most a bounded number of a 's. We can restrict our proof to the single-letter alphabet $\{a\}$, since from any appropriate grammar P for L over the alphabet X we can construct an appropriate grammar P' for L over the alphabet $\{a\}$ with the same (or less) number of productions. We can do this by iteratively removing all

nonterminals other than a , all nonterminals which *only* produce the empty word, and all productions into the empty word.

If P generates L and is in Chomsky normalform, then each syntax-tree T for a^n has depth $\geq \log n$. Along T 's longest path, all productions must be different since if not, then we can copy the fragment in between the occurrences of two equal productions as often as we want, thus producing an unbounded number of a 's. This consideration shows that L has at least $\log n$ many productions. Since any context-free grammar can be transformed into Chomsky normalform with only polynomial blow-up, a $\Omega((\log n)^{\frac{1}{k}})$ bound holds for any grammar of this particular type.

The proof for $L = L_{a,\geq n}$ is basically the same. Here, we can remove the fragment in between the occurrences of two equal productions and thus produce a number of a 's which is strictly smaller than n .

We now give a proof for the existence of unambiguous grammars in Chomsky normalform with $O(\log n)$ many productions. It suffices to prove this for $L = L_{a,=n}, L_{a,\leq n}$ only since

$$L_{a,[n,n+k]} = L_{a,=n-1}aL_{a,\leq k} \quad \text{and} \quad L_{a,\geq n} = L_{a,=n-1}aX^*.$$

This allows to construct appropriate unambiguous grammars for $L = L_{a,[n,n+k]}, L_{a,\geq n}$ by the disjoint union of three unambiguous grammars of size $O(\log n)$ and the introduction of a new initial nonterminal. The unambiguity then follows from the unambiguity of $L_{a,=n-1}$ and, for $w \in X^*$, the uniqueness of a prefix v of w with $v \in L_{a,=n-1}a$.

Further, we will only show the proof for $L(P) = L_{a,\leq n}$ since the construction of the grammar for $L_{a,=n}$ is much easier. We will also restrict our proof to the single-letter alphabet $\{a\}$. Unambiguous grammars for alphabets containing other letters can be constructed by substitution of the terminal letter a by productions for the language $a(X \setminus a)^*$ and then multiplying $(X \setminus a)^*$ with the resulting language.

The proof is adapted from [CeDi 04]. Their idea is to simulate a balanced binary tree. Consider the following set of productions P_0 :

$$V_0 \rightarrow a \quad B_0 \rightarrow a \quad B_1 \rightarrow B_0B_0$$

$$V_h \rightarrow V_{h-1}V_{h-1} \quad (h = 1.. \lfloor \log n \rfloor)$$

and

$$B_h \rightarrow B_{h-1}V_{h-2}|V_{h-1}B_{h-1} \quad (h = 2.. \lfloor \log n \rfloor) \quad .$$

It follows for P_0 :

(i) Each V_h generates exactly one syntax-tree for $a^{(2^h)}$.

(ii) Each B_h generates exactly one syntax-tree for each a^r with $2^{h-1} < r \leq 2^h$.

(i) is obvious. We will prove (ii) by induction. The case $h = 0, 1$ follows from the definition. For $h \geq 2$, the production $B_h \rightarrow B_{h-1}V_{h-2}$ allows to produce exactly one syntax-tree for each a^r with

$2^{h-1} < r \leq 3 \cdot 2^{h-2}$. Similarly for the production $B_h \rightarrow V_{h-1}B_{h-1}$ and $3 \cdot 2^{h-2} + 1 \leq r \leq 2^h$. So both together generate exactly one syntax-tree for each a^r with $2^{h-1} < r \leq 2^h$.

Now choose the maximum $h' \in \mathbb{N}$ with $2^{h'} < n$. Without loss of generality we may assume $n \geq 3$. We introduce the new productions

$$S \rightarrow B_0|B_1|\dots|B_{h'}|R_{h'} \quad (4.4)$$

and the nonterminals $R_{h'}, R_{h'-1}, \dots, R_0$. With the former nonterminals $B_0 \dots B_{h'}$ we can produce exactly one syntax-tree for the strings a^i with $1 \leq i \leq 2^{h'}$. The latter nonterminals R_s shall then produce exactly one syntax-tree for each a^i with $2^s < i \leq k(s) \leq 2^{s+1}$, $s = 0..h'$, where $k(s)$ and the productions for the R_s will be defined below:

We first start with $s = h'$ and set $k(h') := n$. For $s = h', h' - 1, \dots, 1$ we define recursively:

1. **case** $k(s) \leq 2^s + 2^{s-1}$: We set $k(s-1) := k(s) - 2^{s-1}$ and introduce the production $R_s \rightarrow R_{s-1}V_{s-1}$.
2. **case** $k(s) > 2^s + 2^{s-1}$: We set $k(s-1) := k(s) - 2^s$ and introduce the productions $R_s \rightarrow V_s R_{s-1} | B_s V_{s-1}$.

Finally we introduce the production $R_0 \rightarrow V_0 V_0$. All these new productions and P_0 together shall constitute the set P_1 . For P_1 we claim that each R_s produces exactly one syntax-tree for a^i , $2^s < i \leq k(s) \leq 2^{s+1}$. Because of $k(h') = n$ the nonterminal S will then generate all a^i for $i = 1..n$.

For $s = 0$ it is $i = k(s) = 2$ and because $R_0 \rightarrow V_0 V_0$ is the only production with R_0 on the left side the claim holds.

For all $s > 0$ we prove the claim by induction, assuming that it already holds for all smaller s :

1. In case of $k(s) \leq 2^s + 2^{s-1}$: From the inductive assumption it follows that R_{s-1} generates exactly one syntax-tree for each a^i with $2^{s-1} < i \leq k(s) - 2^{s-1}$, while V_{s-1} generates exactly one syntax-tree for $a^{(2^{s-1})}$. Both together prove the claim.
2. In case of $k(s) > 2^s + 2^{s-1}$: According to our inductive assumption, R_{s-1} generates exactly one syntax-tree for each a^i with $2^{s-1} < i \leq k(s) - 2^s$, hence $V_s R_{s-1}$ generates the a^i with $2^s + 2^{s-1} < i \leq k(s)$. The missing a^i with $2^s < i \leq 2^s + 2^{s-1}$ are generated by $B_s V_{s-1}$.

We obtain our grammar by using S as the initial nonterminal and by eliminating the chain productions in 4.4 from P_1 . Since the nonterminals $B_i, R_{h'}$ never occur on the left side of any further chain productions, the resulting grammar will have no more than $6h' + 5 \leq 6(n+1) + 5$ many productions. \square

4.4 Applications

We are now ready to show some applications. We start with counting the number of paths of a particular length (including paths with cycles) and constructing solutions. Further we show that our framework

allows us to find pseudo-polynomial algorithms for NP-hard optimization problems such as the bounded-delay shortest-path and the knapsack problem.

4.4.1 Combinatorial Path Problems

We want to show how the languages from section 4.3.1 can be used in order to solve some combinatorial problems on directed graphs:

Let $G = (N, E)$ be a directed graph, with edges $(s, t) \in E$ labelled with letters $\phi((s, t)) \in \{a, b\}$ and costs $c((s, t)) \leq 0$. Also two natural numbers $m \leq n$ are given, encoded in binary representation. By employing the propositions 18 and 19 and the grammars from section 4.3.1 we can solve the following problems in polynomial time:

1. For all $s, t \in N$, choose a path $\pi : s \rightarrow t$ with minimum costs such that it traverses at least m -many and at most n -many edges labelled with the letter a . Alternatively we may only require that π traverses at least m -many a 's and leave the upper limit open. We can solve this problem in polynomial time since the underlying grammar for $L_{a, [m..n]}$ (or $L_{a, \geq m}$, respectively) can be chosen sufficiently small and such that it is unambiguous (for employing proposition 18), while the underlying semiring is idempotent (for employing proposition 19).
2. For all $s, t \in N$, compute the number of paths $\pi : s \rightarrow t$ which traverse at least m -many and at most n -many edges. Therefore we label all edges with the letter a and choose all edge-costs being 1. We can solve this problem in polynomial time since the underlying grammar can be chosen sufficiently small and such that it is unambiguous (for employing proposition 18). Furthermore we restrict the alphabet to the letter a such that the language is finite (for employing proposition 19). It does not matter that the underlying semiring $(\mathbb{N}, +, \cdot, 0, 1)$ is not idempotent since the generated language is finite.
3. Again, we restrict the alphabet and all edge labels to the single letter a . Interpreting the costs as probabilities from the semiring $([0..1], \max, \cdot, 0, 1)$, choose for all $s, t \in N$ a path $\pi : s \rightarrow t$ with maximum total probability (by multiplying the costs of all its edges) with length $|\pi| \in \{m, \dots, n\}$. We can solve this problem in polynomial time since the underlying grammar can be chosen sufficiently small and such that it is unambiguous (for employing proposition 18) and the language is finite (for employing proposition 19).

Note that the numbers m, n can be exponentially large since they are encoded in binary representation. For representing the paths, we can choose the binary trees from the first chapter.

4.4.2 Transportation Problems

Consider a city with n bus stations $\{1..n\}$ and m bus lines B_j , $j = 1..m$. A line B_j traverses a sequence of stations $s_{j,1}, s_{j,2}, \dots, s_{j,r(j)}$. At each station $i = 1..n$, entering a bus of line j costs $c(i, j) \geq 0$. A group of visitors plans to visit the stations t_1, t_2, \dots, t_z consecutively, in the given order. If possible, they want

to stay in the bus in order to avoid costs for changing the bus. The task is to choose the lines in such way that the total cost of the trip are minimal.

We can describe this problem as a context-free restricted shortest-path problem solvable in polynomial time. The alphabet is the set of all stations $X = \{1..n\}$. The language restriction is given by a context-free grammar for the language $L = t_1 X^* t_2 X^* \dots X^* t_z$, describing all feasible tours. For the set of nodes we choose m disjoint copies of the stations $\{1..n\}$:

$$N = \bigcup_{j=1..m} \{1^{(j)}..n^{(j)}\}$$

For each line B_j we connect the nodes $s_{j,1}^{(j)}, s_{j,2}^{(j)}, \dots, s_{j,r(j)}^{(j)}$ consecutively by edges. An edge $(s_{j,k}^{(j)}, s_{j,k+1}^{(j)})$ is labelled with zero cost (since payment is due only when the bus is entered) and the letter $s_{j,k+1}^{(j)}$ (which is the next station being traversed). Busses of line j are entered via edges of the form $(i, s_{j,k}^{(j)})$ with $i = s_{j,k}$. In this case the cost $c(i, j)$ is due, which becomes the cost label of the edge $(i, s_{j,k}^{(j)})$. As the letter of the same edge we choose the station i . We can always exit a bus via edges of the form $(s_{j,k}^{(j)}, i)$ with $i = s_{j,k}$ for zero cost, hence those edges will be labelled with cost 0 and letter i .

A feasible solution is now any minimum-cost path in $G(N, E)$ with edges and labels as described above and the language-restriction $L = t_1 X^* t_2 X^* \dots X^* t_z$. This problem can be solved by algorithm 4.1.

4.4.3 Algebraic Minimum Syntax-Tree

Consider the a R -valued grammar $P = (X, V, p, S)$ such that R is a bounded-tropical semiring. The task is to compute

$$\min_{w \in X^*} L(P)(w) .$$

We can solve this problem by algorithm 4.1 with P and the following graph $G = (N, E)$: G is a directed graph consisting solely of the single node 1 and $|X|$ multiple edges $(1, 1)$, one for each letter $x \in X$. We do not consider multiple edges for solely technical reasons. Nevertheless one can adapt algorithm 4.1 without any problems.

4.4.4 Deriving Pseudo-polynomial Algorithms

Some hard optimization problems, including many NP-complete ones, allow polynomial algorithms in the case we switch from binary representation of numbers to unitary representation. Then a number $k \in \mathbb{N}$ is represented by k consecutive 1's. This causes an exponential blow up of the input size, which is acceptable if the numbers are small or coarse enough.

The framework presented in this chapter is also useful for a tentative design of pseudo-polynomial algorithms for hard problems. Here we will introduce the knapsack problem and the delay-bounded APSP problem. In both cases, the resulting algorithm is similar to well known pseudo-polynomial algorithms. The difference is the perspective which we take here: But instead of going into the details of the solution method for the respective problem, we will transform our problem to the context-free restricted APSD problem. This happens in such a way that we can apply the algorithm 4.1. The resulting algorithm is

good enough for a tentative approach but generally needs some tweaking in order to be comparative to the known algorithms. None the less it runs in polynomial time.

We start with a less known though important NP-hard problem:

Definition 14 *The **delay-bounded** all-pairs-shortest-path problem consists of a complete directed graph $G = (N, E)$ with positive rational edge costs $c((s, t))$, $s, t \in N$, in binary representation. Additionally the edges are labelled with delays $d((s, t)) > 0$ and a delay bound $D > 0$ is imposed on all feasible paths. The task is to compute, for all pairs $s, t \in N$, their smallest feasible distance*

$$\min_{\substack{\pi: s \rightarrow t \\ d(\pi) \leq D}} c(\pi) .$$

For rational delays and delay-bound, reasonable approximation algorithms have first been proposed by Hassin, see [Has 98]. It often happens that the first polynomial-time algorithms for a suitably modified hard problem gives rise to a series of increasingly better algorithms. We demonstrate that the framework developed in this chapter is suitable for a tentative approach for a pseudo-polynomial algorithm:

Proposition 21 *If delays and delay-bounds $D, d(s, t) \in \mathbb{N}$, $s, t \in N$ are encoded in unitary representation, the delay-bounded APSD problem is solvable in polynomial-time.*

PROOF: Let $G = (N, E)$ be a complete directed graph with edge costs $c((s, t)) > 0$ and delays $d((s, t)) \in \mathbb{N}$, $s, t \in N$, and a delay-bound $D \in \mathbb{N}$, the delays and delay-bounds encoded in unitary representation. We choose the alphabet $A = \{a\}$ and label all edges $(s, t) \in N$ with $\phi((s, t)) = a^{d(s, t)}$.

The original formulation of the context-free restricted APSD problem only allows single letters as labels, but without loss of generality we can always split edges (s, t) labelled with a string $\phi((s, t)) = x_1 \cdots x_k$, $x_i \in X$ into k -many edges, the first one labelled with $c((s, t))$ as costs and x_1 as a letter, and all the others consecutively with zero-costs and the other x_i as letters.

We can now choose the grammar for the language $L_{a, \leq D}$ as a context-free restriction and then use the propositions 18 and 19 in order to solve the context-free restricted APSD problem for the instance G with labels c, ϕ and the language restriction $L_{a, \leq D}$. \square

As a note, for getting a pseudo-polynomial algorithm a simple regular grammar for $L_{a, \leq D}$ is sufficient. However, the actual time bound of the algorithm can be somewhat improved by choosing the grammar from section 4.3.1.

Another example is the knapsack problem:

Definition 15 *An instance of the knapsack problem consists of a series of k items, each one described by its value $c_i > 0$ and its weight $w_i > 0$. There is a weight bound $W > 0$ and the task is to choose $I \subseteq \{1..k\}$ such that $\sum_{i \in I} w_i \leq W$ and $\sum_{i \in I} c_i$ is maximized.*

We give another proof for the following classic result:

Proposition 22 *Given an instance of the knapsack problem $c_i, w_i, W, i = 1..k$ where the weights w_i and their bound W are natural numbers in unitary representation. The costs c_i are allowed to be rational numbers encoded in binary representation. Then the knapsack problem is solvable in polynomial time.*

PROOF: It is not difficult to give a direct proof, but again we want to employ the context-free restricted APSD problem. Let $S = \sum_{i=1}^k w_i$ be the sum of all item weights. If $S \leq W$ we can choose $I = \{1..k\}$ and are ready.

Otherwise, instead of computing the set of items I directly, we compute its complement $J = \{1..k\} \setminus I$. The task is then to minimize $\sum_{i \in J} c_i$ such that $\sum_{i \in J} w_i \geq W - S$. We can transform this into a context-free restricted APSP problem which is, according to propositions 18 and 19, solvable in polynomial-time: The language-restriction is given by a grammar for the language $L_{a, \geq S-W} \subseteq \{a, b\}^*$. The directed graph $G = (N, E)$ is given by the nodes $N = \{0..k\} \cup \{0'..k'\}$ with edges $E = \{(i-1, i) : i = 1..k\} \cup \{(i-1, i') : i = 1..k\} \cup \{(i', i) : i = 1..k\}$. All unions are disjoint. Edges of the form $(i-1, i)$ or (i', i) are labelled with zero costs and with letter b . An edge $(i-1, i'), i = 1..k$ is labelled with cost $c((i-1, i')) = c_i$ and with the string $\phi((i-1, i')) = a^{w_i}$. Again, the loops can be broken up into letter-valued edges, so that we can employ the $L_{a, \geq S-W}$ -restricted shortest-path algorithm. We define J as the set of all i such that the resulting path traverses i' . This indicates that item i is subtracted from the collection i of all items. \square

4.5 Related Work

The semiring framework presented in this chapter is new in the context of language-restricted path-problems. A recent paper on language-restricted path problems is [BJM 98]. Here, many hardness proofs for subproblems are presented as well as further applications. Our algorithm 4.1 is a generalization of the dynamic programming scheme represented there. The algebraic minimum syntax-tree problem is related to Knuth's grammar problem, see [Knu 77]. Sadly, Knuth's superior functions cannot be cast into the semiring framework, but the resulting dynamic programming scheme and its proof of correctness resemble that of algorithm 4.1.

Chapter 5

Results from Formal Language Theory

The algorithms for the context-free language-restricted-all-pairs-shortest-path problem and for path-parsing from the previous chapter are considerably general. For many instances of semirings those algorithms run in polynomial time.

Two questions arise: Firstly, do related problems exist which are more difficult than path-parsing? Secondly, for which classes of context-free languages can we expect to increase runtime efficiency of the algorithm?

In the first section of this chapter we discuss the upper limits of language restricted path problems. In particular we will show that certain permutations of feasible solutions cause very simple membership problems to be NP-hard. The second section shows for idempotent semirings that associativity of the multiplication in the CYK-algorithm coincides with regularity of the language itself. This diminishes our hope of exploiting associativity for simplifying our algorithms while still retaining more power than that of regular languages.

Finally, the observation that determinism may give rise to faster algorithms motivates us to find tests whether a given language is deterministic context-free. We present a corrected version of the KC-DCFL-lemma by Vitanyi and Lie which helps proving that some context-free languages are non-deterministic context-free.

5.1 Permutations of Strings

We have seen in the fourth chapter how semirings can be used in order to generalize parsing and path problems. The APMS problem and the language-restricted path problem give a versatile model for many related combinatorial problems. Unfortunately, context-free languages have several restrictions. For instance, it is well-known that the class of context-free languages is not closed under intersection. In case of the context-free language-restricted-path problem, this disallows to combine two arbitrary language restrictions in such a way that both have to be satisfied by feasible solutions.

It is not the purpose of this section to recall closure-properties of context-free languages or to generalize their properties for semiring-valued languages. The interested reader may consult [KuSa 85], which

gives an extensive treatment of the connections between semirings and formal languages.¹

Instead we will deal with another restriction of context-free languages in connection to permutations of string fragments: We have seen that several generalizations of the shortest-path problem can be modelled by semiring-valued context-free grammars and that polynomial-time algorithms do exist. Often, part of a problem instance is a sequence of destinations which must be visited in a previously given order. If we give up this restriction and allow arbitrary order, we get the travelling salesman problem — which is *NP*-hard. As a first remark, it thus becomes clear that shortest-simple-path-language-restricted is *NP*-hard already for regular languages since for any graph G , we can reduce HAMILTONIANPATH of G to a simple path problem of an appropriately labeled graph G' , restricted by the regular language which ensures that each node is traversed at least once. Much deeper results with respect to fixed regular and context-free languages and specialized graph classes can be found in [BCDMS 99].

The introduction of arbitrary permutations often results in infeasible (in the sense of *NP*-hard) problems. We will briefly discuss the implications of this observation with respect to language recognition.

Let $L \subseteq X^*$ be an arbitrary language over the alphabet X . We write $L \in P$ if its corresponding membership problem is decidable in polynomial time, and similarly for other complexity classes. The group of permutations acting on sets of size n is denoted by S_n .

Definition 16 For $f, g : \mathbb{N} \rightarrow \mathbb{N}$, we define a f, g -permutation operator on languages:

$$\begin{aligned} \text{perm}(f, g, L) := \{ & x_1 \cdots x_{f(n)} : n \geq 0 \quad \wedge \quad \forall i = 1..f(n). |x_i| = g(n) \\ & \wedge \quad \exists \pi \in S_{f(n)}. x_{\pi(1)} \cdots x_{\pi(f(n))} \in L \} \end{aligned}$$

for $L \subseteq X^*$.

Furthermore, for a class of languages \mathcal{L} and sets of functions $F, G \subseteq \mathbb{N}^{\mathbb{N}}$ we set

$$\text{perm}(F, G, \mathcal{L}) := \{ \text{perm}(f, g, L) : f \in F, g \in G, L \in \mathcal{L} \} \quad .$$

Informally stated, $\text{perm}(f, g, L)$ takes, for each $n \geq 0$, all strings $w \in L$ with $|w| = f(n)g(n)$ and then cuts w into $f(n)$ pieces of equal length, which are then rearranged in arbitrary order.

Since f and g can cause $\text{perm}(f, g, L)$ to do rather wild things with a language L , the classes of context-

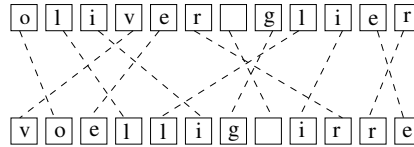


Figure 5.1: A permutation of the author's name with $f = 12$ and $g = 1$.

free and regular languages are often not closed under interesting classes of permutations. However, one

¹In [KuSa 85] semiring-valued languages are instances of formal power series over semirings. Their algebraic framework allows to give unified proofs for many results from formal language theory.

can ask under which types of permutations a given language is still efficiently decidable. For instance we have

$$\text{perm}(O(1), \text{poly}, \mathcal{L}_{cf}) \subseteq P \quad ,$$

where poly denotes the polynomial functions over a single variable and \mathcal{L}_{cf} is the class of context-free languages. The observation is due to the fact that for any $L \in \mathcal{L}_{cf}$ we can decide the membership problem in polynomial time. Assume that L , a constant c , and a polynomial function p are fixed. Now, given a string w of length $|w| = cp(n)$, $n \geq 0$, we can decide whether $w \in \text{perm}(c, p, L)$ by testing membership in L for only c many permutations of w . On the other hand, we easily get NP-complete problems already for $f = O(n)$:

Proposition 23 *It is $\text{perm}(\text{poly}, \text{poly}, \mathcal{L}_{cf}) \subseteq NP$ and there exists a NP-complete language in $\text{perm}(n, n, \mathcal{L}_{cf})$.*

PROOF: To see that $\text{perm}(f, g, L) \in NP$ for polynomial functions f, g and $L \in \mathcal{L}_{cf}$ one needs to check all appropriate permutations of a given input string by a pushdown-automaton.

The proof of the second part of the proposition is by reduction of HAMILTONIANPATH, which is the problem of deciding whether a connected undirected graph $G = (N, E)$ with n nodes has a path which visits each node exactly once. Let $X = \{0, 1, a, b\}$ be an alphabet of four distinct letters. We use 0, 1 in order to assign a unique codeword $C(p)$ of length $k = \lceil \log |N| \rceil$ to each node $p \in N$. Now, for each $p \in N$, let $B(p)$ be a string of length $n(k+1)$ of the form

$$B(p) = C(p)^{-1} a x_1 b x_2 b \cdots b x_{n-1} b$$

with $x_i \in \{0, 1\}^k$ such that a string $v \in \{0, 1\}^k$ is an infix of $B(p)$ if and only if there exists a node $q \neq p$ adjacent to p with $C(q) = w$. Note that all $B(p)$ have the same length which we denote by $r(G)$. In order to satisfy the length condition, a code $C(q)$ may appear multiple times in $B(p)$. A graph $G = (N, E)$ with nodes $N = (p_1, \dots, p_n)$ is then represented by

$$B(G) = B(p_1)B(p_2) \cdots B(p_n) \quad ,$$

such a representation can be efficiently obtained from any “natural” representation of G . The language

$$L = \{0, 1\}^* a \{uwbvbw^{-1}a : w \in \{0, 1\}^*, uv \in \{0, 1, b\}^*\}^* \{0, 1, b\}^* \quad (5.1)$$

is obviously a context-free language. Now, G has a Hamiltonian path if and only if $B(G)b^{r(G)^2 - nr(G)} \in \text{perm}(n, n, L)$. \square

By using the same encoding of a graph as in the proof above, we get the following corollary:

Corollary 24 *The following problem is NP-complete: Given a string $w \in X^*$ and a regular grammar P which generates a finite language $L(P) \subseteq X^*$, decide whether $w \in \text{perm}(n, n, L(P))$.*

PROOF: The proof is, for a given graph G with n nodes, by replacing the middle part of the language L in 5.1 by the language

$$\left\{ uwbvbw^{-1}a : w \in \{0, 1\}^k, uv \in \{0, 1, b\}^*, |wbvbw^{-1}a| = r(G) \right\}$$

and then taking it to the power $r(G)$. Since there are only $O(n)$ possible codes $w = C(p)$ of length $k = \lceil \log |N| \rceil$, the result is a finite language

$$L' = \{0,1\}^* a \left\{ uwbvbw^{-1}a : w \in \{0,1\}^k, uv \in \{0,1,b\}^*, |wbvbw^{-1}a| = r(G) \right\}^{r(G)} \{0,1,b\}^*$$

which can be represented by a regular grammar of polynomial size. Now, G has a Hamiltonian path if and only if $B(G)b^{r(G)^2 - nr(G)} \in \text{perm}(n, n, L')$. \square

5.2 Associativity and Regularity

Throughout this section, let $R = (R, +, \times, 0, 1)$ be an idempotent semiring, i.e. $a + a = a$ for all $a \in R$ holds. Given a context-free R -valued grammar $P = (X, V, p, S)$ in Chomsky normalform, we define the **CYK-product** $\otimes_P : R^V \times R^V \rightarrow R^V$ of two maps $f : V \rightarrow R, g : V \rightarrow R$ as

$$(f \otimes_P g)(A) := \sum_{B, C \in V} p(A \rightarrow BC) \times f(B) \times g(C)$$

for $A \in V$. The CYK-product \otimes_P distributes over componentwise sums on R^V , and we say that a grammar P is **associative** if P is context-free in Chomsky normalform and its CYK-product is associative.

The CYK-parsing algorithm and the recursion formula 4.3 for the APMS problem could be simplified in those cases in which \otimes_P is associative. In particular, for idempotent semiring parsing of a string $w = x_1 \cdots x_n$ with letters $x_i \in X$ we get

$$L(P)(w) = (f_1 \otimes_P (f_2 \otimes_P \cdots (f_{n-1} \otimes_P f_n) \cdots))(S), \quad (5.2)$$

where for $1 \leq i \leq n$ the function $f_i : V \rightarrow R$ maps a nonterminal A to the value $p(A \rightarrow x_i)$.

Unfortunately, context-free grammars are not associative in general. Equation 5.2 indeed suggests that associative grammars generate languages which must be considerably simpler than general context-free R -valued languages. We will see that for idempotent R , the languages which are generated by an associative grammar are exactly the R -regular languages.

A **non-deterministic R -valued finite state automaton** is a tuple $M = (X, Z, \delta, \hat{s}, F)$, where X is M 's finite input alphabet, Z is its finite set of states, \hat{s} is the initial state and $F \subseteq Z$ are the final states, and $\delta : Z \times Z \times X \rightarrow R$ is the R -valued transition relation of M . The automaton M gives rise to an R -valued language $L(M) : X^* \rightarrow R$, where for $w \in X^*$:

$$L(M)(w) = \sum_{\perp \in F} \sum_{\substack{\pi: \hat{s} \rightarrow \perp \\ |\pi|=w}} \|\pi\|_{\delta, w}.$$

Here, $|\pi|$ is the number of traversed edges of a path $\pi = \pi_0 \dots \pi_n, \pi_i \in Z$ in the complete graph $(Z, Z \times Z)$, and $\|\pi\|_{\delta, w}$ is the product of R -values for the string $w = x_1 \dots x_n, x_i \in X$, along π :

$$\|\pi\|_{\delta, w} := \delta(\pi_0, \pi_1, w_1) \times \cdots \times \delta(\pi_{n-1}, \pi_n, w_n)$$

Finally, we say that $L : X^* \rightarrow R$ is **R -regular** (or simply: regular) if there exists a R -valued non-deterministic automaton with $L(M) = L$.

Proposition 25 *Let R be an idempotent semiring. A context-free R -valued language $L : X^* \rightarrow R$ with $L(\varepsilon) = 0$ is regular if and only if there exists an associative context-free R -valued grammar P such that $L = L(P)$.*

PROOF: Assume that L is regular, i.e. there exists a R -valued non-deterministic finite state automaton $M = (X, Z, \delta, \hat{s}, F)$ with $L(M) = L$. We construct a grammar $P = (X, V, p, S)$ which allows to generate an arbitrary path π of length n in the complete graph $(Z, Z \times Z)$ and then evaluates $\|\pi\|_{\delta, w}$ for a given string w of length $n \geq 2$. Therefore we set $V = (Z \times Z) \cup \{S\}$ for some $S \notin (Z \times Z) \cup X$. Now we define $p : V^3 \cup (V \times X) \rightarrow R$. All productions with S on the right side are given the value 0:

$$p(A \rightarrow BS) := p(A \rightarrow SB) := 0$$

for all $A, B \in V$. In order to take care of single-letter strings $x \in X$, we set

$$p(S \rightarrow x) := \sum_{\perp \in F} \delta(\hat{s}, \perp, x)$$

for all $x \in X$. In order to generate paths in $(Z, Z \times Z)$, we further set

$$p((s_1, s_2) \rightarrow (r_1, r_2)(t_1, t_2)) := \begin{cases} 1, & \text{if } s_1 = r_1 \wedge s_2 = t_2 \wedge r_2 = t_1 \\ 0, & \text{else} \end{cases}$$

and

$$p(S \rightarrow (s_1, s_2)(t_1, t_2)) := \begin{cases} 1, & \text{if } s_1 = \hat{s} \wedge t_2 \in F \wedge s_2 = t_1 \\ 0, & \text{else} \end{cases}$$

for $s_1, s_2, r_1, r_2, t_1, t_2 \in Z$. Finally we must give values for productions $(s_1, s_2) \rightarrow x$ for $s_1, s_2 \in Z$ and $x \in X$:

$$p((s_1, s_2) \rightarrow x) := \delta(s_1, s_2, x)$$

We sketch now how to prove correctness of our construction, i.e. that indeed $L(P) = L(M)$: Since $+$ is idempotent, it suffices to show that the following equality between sets

$$\{\|\pi\|_{\delta, w} \in R : |\pi| = |w| \wedge \exists \perp \in F. \pi : \hat{s} \rightarrow f\} \cup 0 = \{p(T) : T \text{ is a syntax-tree with } S \Rightarrow_T^* w\} \cup 0$$

holds for each $w \in X^+$ (the extension of p to syntax-trees has been introduced in section 4.1). Equality of both sets can then be shown by the following observation for each potential syntax-tree T for $S \Rightarrow_T^* w$: For non-zero $p(T)$, T must produce a valid path in the complete graph π in $(Z, Z \times Z)$. Then T transforms the edges of π of the form $(s_1, s_2) \in Z \times Z$ into w 's letters $x \in X$, while the evaluation $p(T)$ multiplies the values of the form $\delta(s_1, s_2, x)$ in the appropriate order. Furthermore, such a syntax tree exists for each appropriate path for w .

Finally we can verify that \otimes_P is associative by observing that for arbitrary $f, g, h : V \rightarrow R$ and $A \in V$ the expressions $((f \otimes_P g) \otimes_P h)(A)$ and $(f \otimes_P (g \otimes_P h))(A)$ both evaluate to

$$\sum_{B, C, D, E \in V} p(A \rightarrow BC) \times p(C \rightarrow DE) \times f(D) \times g(E) \times h(C). \quad (5.3)$$

For $((f \otimes_P g) \otimes_P h)(A)$ we get the expression 5.3 directly, while for $(f \otimes_P (g \otimes_P h))(A)$ we can rearrange the factors accordingly since $p(A \rightarrow BC), p(C \rightarrow DE) \in \{0, 1\}$.

For the proof of the opposite direction of proposition 25, we assume that $L : X^* \rightarrow R$ is determined by an associative context-free grammar $P = (X, V, p, S)$ with $L(P) = L$. We want to construct an R -valued non-deterministic finite automaton $M = (X, Z, \delta, \hat{s}, F)$ such that $L(M) = L(P)$. Therefore we set $Z = V \cup \{\perp\}$ for some additional state $\perp \notin V$. We take $\hat{s} = S$ as the initial state of M and $F = \{\perp\}$ as the only final state. In order to take care of single-letter strings and termination, we set

$$\delta(A, \perp, x) := p(A \rightarrow x)$$

for all $x \in X$ and $A \in V$.

Since \perp is not a nonterminal of P and hence cannot appear on a right side of a production rule, we set

$$\delta(\perp, A, x) := 0$$

for all $x \in X$ and $A \in V$. For all other states $A, B \in V = Z \setminus \{\perp\}$, we set

$$\delta(A, B, x) := \sum_{C \in V} p(A \rightarrow BC) \times p(C \rightarrow x)$$

for all $x \in X$.

According to these definitions, we get by using idempotency

$$\begin{aligned} L(M)(w) &= \sum_{\substack{A_0, \dots, A_n \in V \\ A_0 = S}} \left(\prod_{i=1}^{n-1} \sum_{C \in V} p(A_{i-1} \rightarrow A_i C) \times p(C \rightarrow x_i) \right) \times p(A_n \rightarrow x_n) \\ &= \sum_{\substack{A_0, \dots, A_n, C_1, \dots, C_{n-1} \in V \\ A_0 = S}} \left(\prod_{i=1}^{n-1} p(A_{i-1} \rightarrow A_i C_i) \times p(C_i \rightarrow x_i) \right) \times p(A_n \rightarrow x_n) \end{aligned}$$

for each $w = x_1 \cdots x_n \in X^+$ with letters $x_i \in X$. Similarly we get for equation 5.2:

$$\begin{aligned} L(P)(w) &= (f_1 \otimes_P (f_2 \otimes_P \cdots (f_{n-1} \otimes_P f_n) \cdots))(S) \\ &= \sum_{A_1, C_1 \in V} p(S \rightarrow A_1 C_1) \times p(A_1 \rightarrow x_1) \times (f_2 \otimes_P \cdots (f_{n-1} \otimes_P f_n) \cdots)(C_1) \\ &= \sum_{A_1, C_1 \in V} p(S \rightarrow A_1 C_1) \times p(A_1 \rightarrow x_1) \times \left(\sum_{A_2, C_2 \in V} p(A_1 \rightarrow A_2 C_2) \times p(A_2 \rightarrow x_2) \times \cdots \right. \\ &\quad \left. \left(\sum_{A_{n-1}, C_{n-1} \in V} p(A_{n-2} \rightarrow A_{n-1} A_n) \times p(C_{n-1} \rightarrow x_{n-1}) \times f_n(A_n) \right) \cdots \right) \\ &= \sum_{\substack{A_0, \dots, A_n, C_1, \dots, C_{n-1} \in V \\ A_0 = S}} \left(\prod_{i=1}^{n-1} p(A_{i-1} \rightarrow A_i C_i) \times p(C_i \rightarrow x_i) \right) \times p(A_n \rightarrow x_n) \end{aligned}$$

This proves that $L(M) = L(P)$. □

Proposition 25 is interesting for two reasons: First, it follows as a corollary that if $L : X^* \rightarrow R$ is generated by an associative grammar, then its non-zero support $L' = \{w \in X^* : L(w) \neq 0\}$ is a regular language $L' \subseteq X^*$. It also shows that associativity does not help much in simplifying algorithms for general semiring parsing and the APMS problem since in the idempotent case, we can make better use of the regularity than by utilizing associativity. Secondly, proposition 25 gives us an alternative definition of R -regular languages in case of an idempotent semiring R , especially in the boolean case and for tropical semirings. It does not, however, hold for the ring of natural numbers². If in those cases we utilize associativity, we should think first what we actually want to count with our given semiring. However, there seems to be little practical value for counting in a non-idempotent semiring such as the ring of natural numbers by using an associative context-free grammar.

5.3 Kolmogorov Complexity and Deterministic Context-Freeness

In the fourth chapter we have seen that unambiguous grammars play a special role in reducing language-restricted-path-problems to path-parsing. Fortunately, the class of unambiguous context-free languages is large, in the sense that it contains important classes of context-free languages such as the class of deterministic context-free languages. Also, for special graphs — such as rooted trees — one might also devise specialized algorithms based on deterministic parsing.

As such, a first test of a given language restriction should be whether or not the underlying language is deterministic context-free. Sadly we do not know any non-trivial characterisation of deterministic context-free languages, which makes it easy to prove that a language is not deterministic context-free. While for the positive case it suffices to construct an appropriate pushdown-automaton or grammar, the difficulty usually lies in showing that a given language is *not* deterministic context-free. In this section we give a criterion which is in many cases sufficient to prove that a given context-free language L is inherently not deterministic context-free. The criterion is based on Kolmogorov complexity.

5.3.1 A Short Introduction into Kolmogorov Complexity

Let X be an alphabet with at least two distinct symbols 0 and 1. For a string $x = x_1 \dots x_n \in X^*$ we define

$$x^\$:= 1^{\text{BIN}(n)} 0 \text{BIN}(n) x$$

as the **self-delimiting code** for x , where $\text{BIN}(n)$ is the binary representation of the number n . In the sequel, let U be a universal machine which expects some input of the form $p^\$q$, $p, q \in X^*$ and then simulates the program p on input q . Now we define for $x, q \in X^*$:

$$C(x|q) := \min\{|p| : U(p^\$q) = x, p \in X^*\}$$

²The author actually conjectures that for all real-valued languages $L(P), L(M) : X^* \rightarrow \mathbb{R}$ with infinite non-zero support, where $L(P)$ is generated by an associative grammar and $L(M)$ by a finite real-valued automaton, we have $L(P) \neq L(M)$.

is the **conditional Kolmogorov complexity** of x with the additional information q , and

$$C(x) := \min\{|p| : U(p^\$) = x, p \in X^*\}$$

is the **Kolmogorov complexity** of x (without additional information).

The following immediate consequences are stated without proof:

Proposition 26

(i) $C(x|q) \leq C(x) \leq |x| + O(1)$.

(ii) For any totally recursive function $f : X^* \rightarrow X^*$, it is

$$C(f(x)|q) \leq C(x|q) + O(1) \quad .$$

(iii) $C(x) \leq C(q) + C(x|q) + O(\min(\log |x|, \log |q|))$

□

For any natural number $n = 0, 1, 2, \dots$, we denote by \underline{n} the n th string of X^* in lexicographic order. Further we set $\lg(n) := |\underline{n}|$. Informally, the following proposition states that any sufficiently large number m can be enclosed by the length of a number r and r 's much smaller Kolmogorov complexity:

Proposition 27 For sufficiently large m , there always exists an $r \in \mathbb{N}$ with $\lg(r) > m$ and $C(\underline{r}) < \lg(\lg(m))$.

PROOF: We choose $t = \lg(\lg(m))$ and $r = f(f(f(t)))$, where $f : \mathbb{N} \rightarrow \mathbb{N}$ maps a number n to the (unique) number n' such that $\underline{n'}$ is the string 1^{n+1} . By applying proposition 26 (ii) we get

$$C(\underline{r}) \leq C(\underline{t}) + O(1) \leq \lg(t) + O(1) < t = \lg(\lg(m))$$

for sufficiently large m . On the other hand, if m is large enough, it is $f(\lg(m)) > m$ and $\lg(f(m)) > m$. It follows $\lg(r) > m$. □

We say that a string $x \in X^*$ is **compressible** iff $C(x) < |x|$, otherwise it is **incompressible**. Similarly, a number $n \in \mathbb{N}$ is compressible if the n th string \underline{n} in the lexicographic enumeration is compressible. For $n \in \mathbb{N}$ there are only $2^n - 1$ many strings of length shorter than n . Therefore:

Theorem 28 (Incompressibility Theorem) For each $n \in \mathbb{N}$ there exists at least one incompressible string of length n . Hence, there exist infinitely many incompressible strings.

5.3.2 The KC-DCF Lemma

For a language $L \subseteq X^*$ and a string $w \in X^*$, we denote by $x^{-1}L := \{v \in X : vx \in L\}$ the set of strings v that extend x to a string in L . A context-free language $L \subseteq X^*$ is **deterministic context-free** if there exists a deterministic pushdown automaton (dpda) that recognizes L . In order to make the following constructions feasible, we assume without loss of generality that A is *loop-free* (for more details see [Har 78]). Such a dpda can be implemented by a random access machine and thus recognizes L in linear time. Pumping lemmas for proving that a language is not deterministic context-free exist but are difficult to handle. In [LiVi95] the authors use an incompressibility argument in order to give a more intuitive, necessary condition for deterministic context-free languages. Sadly, the formulation and the proof of the KC-DCF lemma contains some mistake which also affects the corollary and hence makes them both wrong. Before we give an alternative formulation for both, the lemma and its corollary, we will discuss the original formulation of the corollary. By giving a counter-example, we disprove the original corollary and — consequently — the original KC-DCF lemma:

Let $x, y \in X^*$ and c be any constant, and let ω be a recursive sequence over X^* . The idea is to repeat y in the input of a dpda that recognizes L , while limiting the amount of information that the dpda keeps on its stack. Eventually, too much information is lost in order to properly unwind the stack. Li/Vitányi's corollary of the KC-DCF lemma states the following: There exists a constant c' such that, for $u, v, w \in X^*$ where u is a suffix of the left-infinite string $\dots y y x$ and v is a prefix of ω , the following three conditions imply $C(w) \leq c'$:

- (i) $C(v|p_{uv'}) \leq c$ for all prefixes v' of v and programs $p_{uv'}$ that list $(uv')^{-1}L$ in lexicographic order;
- (ii) $C(w|p_{uv}) \leq c$ for all programs p_{uv} that list $(uv)^{-1}L$ in lexicographic order;
- (iii) $C(v) \geq 2 \cdot \lg(\lg(|u|))$.

In [LiVi95], condition (i) is restricted to the only prefix $v' = \varepsilon$ as the empty string, omitting all other prefixes of v . But then the corollary is not a direct consequence of the original KC-DCF lemma anymore, since it will not suffice in order to reconstruct all required configurations of the dpda in the corresponding condition of the lemma. However, this is not the essential mistake in [LiVi95] (which we point out later in our proof of the KC-DCF lemma). Since the small correction of condition (i) weakens the corollary to a statement which is properly derivable from the original lemma, the following counter-example disproves them all: the original KC-DCF lemma and the original corollary, with or without the correction.

We show now that there exist deterministic context-free languages L and $u, v, w \in X^*$ satisfying (i),(ii),(iii) but $C(w) \geq c'$ for any constant c' :

Example 2 Set $u = 0^{2n}$, $v = 1^n$, $w = 0^n$, and $L = \{0^n 1^m 0^k : n = m + k\}$. Now, for all prefixes v' of v , v' is the first half of the lexicographically first string in $(uv')^{-1}L$ which is all 1, while w is the lexicographically first string in $(uv)^{-1}L$ which is all 0. Thus (i) and (ii) are satisfied. Now let $c' > 0$ be any constant and choose an incompressible n with $C(w) > c'$. If n is sufficiently large we get $C(v) > \log |u|$ and therefore (iii). However, L is obviously deterministic context-free.

From now on we assume that $L \subseteq X^*$ is an arbitrary nontrivial deterministic context-free language, and A is a dpda recognizing L . We will keep track of the whole configuration of the dpda while processing the input string. A **dpda configuration** is a triple (w, q, k) where w is the current (rest of) the input, q is the current state, and k is a string that represents the stack. On input w , A starts with configuration (w, q_0, ϵ) . The transition function of A is denoted by δ_A , which maps a configuration (w, q, k) to its direct successor configuration.

Now let $w \in X^+$ be an arbitrary input string and i be the maximum i with $\delta_A^i(w, q_0, \epsilon) = (w', q, k)$, $\delta_A^{i+1}(w, q_0, \epsilon) = (w'', q', k')$ and $|w''| < |w'|$. Since L is nontrivial, i exists. So (w', q, k) is the last configuration in which A reads a symbol from the input w , while (w'', q', k') is its direct successor. For any input w , we define $k(w) := k'$ and $q(w) := q'$. Furthermore, for $n \leq |k(w)|$ let $k(w, n)$ be the n topmost symbols of the stack after processing w , ie. $k(w, n) := k(w)_1 \dots k(w)_n$. Our acceptance criterion is the following: For $w \in X^*$, the dpda A accepts w if there exist $u, v \in X^*$ with $w = uv$ if and only if there exists an $i \geq 0$ such that $\delta_A^i(u, q(u), k(u)) = (\epsilon, q, u)$ where q is an accepting state of A .

We will follow the lines of [LiVi95] and observe the behavior of A on all suffixes u of a string of the form $yy \dots yx$. The repetition of y in the input will force a regular structure of the stack's content:

Proposition 29 Let $x, y \in X^*$. Then there exists a constant $c_0 \in \mathbb{N}$ such that the following proposition holds: Let u be a suffix of $yy \dots yx$ with $|k(u)| \geq c_0$ and $r \in \mathbb{N}$. Then there exists a number $s \leq c_0$, such that for the suffix u' of $yy \dots yx$ with length $|u'| = r + s$ the condition

$$q(u) = q(u') \quad \text{and} \quad k(u, |k(u)| - c_0) = k(u', |k(u)| - c_0)$$

is satisfied. That is, after processing the inputs u and u' respectively, the state of A and the $|k(u)| - c_0$ topmost stack symbols are each one the same.

PROOF: We consider finitely many cases for u , and give a sufficient bound for c_0 in each case: Let u be a suffix of x , ie u has the form $x_i x_{i+1} \dots x_{|x|}$. For all finitely many suffixes u of this form we can choose $c_0 > |k(u)|$.

In all other cases, u has the form $u = y_r \dots y_{|y|} y^s x$. We observe A on the infinite input $y_r \dots y_{|y|} yy \dots$ and set

$$(y_{n(i)} \dots y_{|y|} yy \dots, q_i, k_i) = \delta_A^i(y_r \dots y_{|y|} yy \dots, q_0, \epsilon) \quad .$$

If now the size of the stack during processing $y_r \dots y_{|y|} yy \dots$ is bounded, then the set $K = \{k(y_r \dots y_{|y|} y^n x) : n \in \mathbb{N}\}$ is finite and we can choose $c_0 > \max\{|k| : k \in K\}$.

Otherwise assume that the stack k_i becomes arbitrarily large during processing the infinite input

$y_r \dots y_{|y|} y y \dots$. Then the dpda A can only remove a bounded number of symbols from the stack, in the sense that for each n , there exists an i after which the stack's size remains larger than n . Otherwise some triple $(n(i), q_i, k_i)$ will be periodically repeated which in turn implies that the stack size is bounded, in contrast to our assumption.

So for each n there exists a configuration $\delta_A^i(y_r \dots y_{|y|} y y \dots, q_0, \varepsilon)$ such that the size of the stack remains $\geq n$ for all subsequent configurations and all letters below cannot be accessed furthermore. However, if after the i th step the single topmost stack symbol is not accessed anymore, all subsequent configurations only depend on the current state q_i and the current position $n(i)$ in the input's current y . Since there are only finitely many such pairs $(q_i, n(i))$, they must be equal for two $i_1 < i_2$. If we observe the subsequent behavior of A , the dpda will run through the same sequence of operations and states, and thus add the same sequence of symbols to the stack. Because of $n(i_1) = n(i_2)$, the number of steps that A performs inbetween i_1 and i_2 must be a multiple of $|y|$, say $a|y| = i_2 - i_1$. Since the stack grows and never shrinks below its size at step i_1 , a string $w \neq \varepsilon$ will be appended during those subsequent steps. Formally, there exist strings v, w, z and $a, m \in \mathbb{N}$ such that

$$k(y_r \dots y_{|y|} y^m y^{an}) = z w^n w \quad \text{and} \quad q(y_r \dots y_{|y|} y^m y^{an}) = q(y_r \dots y_{|y|} y^m y^{(n+1)a})$$

for each $n \geq 0$. Note that in our notation z 's first character is on the top of the stack. $y_r \dots y_{|y|} y^m$ denotes the processed part of the input until step i_1 . Now we can choose c_0 sufficiently large: First we choose $c_0 > |y^m y^a|$ which will give us a bound with respect to s . Secondly, in order to account on $k(u)$, we treat the finitely many cases for each $j \in \mathbb{N}$ with $1 \leq j \leq a$ separately: We fix such a j and set $u(n) = y_r \dots y_{|y|} y^{m+an+j} x$ for all $n \geq 0$. It may well happen that during processing the x -part of input $u(n)$, the dpda A removes an unlimited number of symbols from the stack. However, if this is the case, then for sufficiently large n the whole stack will be reduced to a suffix of v , because the sequence of states will necessarily repeat during the removal of w^n . Thus, $K = \{k(u(n)) : n \in \mathbb{N}\}$ is finite and we can choose $c_0 > \max\{|k| : k \in K\}$ for input $u(n) = y_r \dots y_{|y|} y^{m+an+j} x$.

On the other hand, we assume that on all inputs $u(n) = y_r \dots y_{|y|} y^{m+an+j} x$, $n \geq 0$, during processing of the x -part no more than c many symbols will be removed from the stack $k(u(n)) = k(y_r \dots y_{|y|} y^m y^{an} y^j) = z w^n v$. Choose $n' \geq 0$ such that $|w^{n'}| > c$. It follows $k(u(n), |k(u(n))| - |v|) = k(u(n+1), |k(u(n))| - |v|)$ and $q(u(n)) = q(u(n+1))$ for all $n > n'$. That is, after processing $u(n)$ or $u(n+1)$ respectively, the $|k(u(n))| - |v|$ topmost stack symbols and the final states are mutually equal. So for all $n > n'$ we can choose $c_0 > |v|$, while for the finitely many $n \leq n'$, we choose $c_0 > \max\{|k(u(n))| : n \leq n'\}$. \square

We will now directly proceed with a corrected version of the KC-DCF lemma. For $v, v_1, v_2 \in X^*$, we call v_1, v_2 a **decompositon** of v iff $v_1 v_2 = v$.

Proposition 30 (KC-DCF-Lemma) *Let $x, y \in X^*$ and c be a constant. Then there exists a constant c' with the following property: Let $u, v, w \in X^*$, where u is a suffix of $yy \dots yx$. Then the following three conditions imply $C(w) \leq c'$:*

(i) $C(v''|k^s q) \leq c$, for all decompositions $v'v'' = v$. Here, k, q is an arbitrary pair of a stack and a state which, on processing v'' , induces the same behavior of A as the pair $k(uv'), q(uv')$;

(ii) $C(w|k(uv)^s q(uv)) \leq c$;

(iii) $C(v) \geq 2 \cdot \lg(\lg(|u|))$.

In (i), same behavior of A for k, u and $k(uv'), q(uv')$ means the same sequence of states and operations. Formally: Let $\text{pr}_{12}(w, q, k) := (w, q)$ be the function that takes a configuration of A and forgets the stack. Then all we assume from k, q is:

$$\forall i \geq 0. \text{pr}_{12}(\delta_A^i(v'', q, k)) = \text{pr}_{12}(\delta_A^i(v'', q(uv'), k(uv'))) \quad .$$

PROOF: of the KC-DCF lemma: Let c_0 be the constant from Proposition 29 and w be fixed. We distinguish two cases:

Case 1: Assume that for all but finitely many pairs (u, v) which satisfy (i)-(iii), the size of the stack shrinks below c_0 during processing the v -part on input uv . Then for each such pair a step i exists with

$$\delta_A^i((v, q(u), k(u))) = (v'', q, k) \quad \text{and} \quad |k| \leq m \quad ,$$

where $v = v'v''$ for some v' and $m \geq c_0$ and where m bounds the stack size for the finitely many remaining pairs (u, v) . Because of (i) we can reconstruct v'' with only c many additional symbols, assuming a description of q and k . Because $|k| \leq m$, this can be done with finitely many symbols.

Knowing v'' and q, k we can reconstruct $q(uv'v'') = q(uv)$ and $k(uv'v'') = k(uv)$ by observing A on input v'' , starting with $q(uv'), k(uv')$. Because of (ii) only c many further symbols will suffice to reconstruct w . Hence $C(w)$ is bounded by a sufficiently large constant c' .

Case 2: Now we show that case 1 always holds. Assume for the contrary, that for infinitely many pairs (u, v)

$$\forall i \geq 0. \quad \delta_A^i((v, q(u), k(u))) = (v'', q, k) \quad \Rightarrow \quad |k| > c_0. \quad (5.4)$$

holds and then derive a contradiction.

For each u there exist only finitely many candidates v which satisfy (i). This is easy to see since in (i), setting $k = k(u)$, $q = q(u)$, and $v' = \varepsilon$ suffices to show that for any given u , the description size of all possible v is bounded. So if we assume there are infinitely many pairs satisfying (5.4), then we can choose arbitrarily long u . Using Proposition 27 we choose $|u|$ large enough such that there exists a number $r \in \mathbb{N}$ with $\lg(r) > |u|$ and

$$C(r) < \lg(\lg(|u|)) \quad .$$

Using Proposition 29 we can find $s \leq c_0$ such that the suffix u' of length $|u'| = r + s$ from $yy \dots yx$ satisfies

$$q(u) = q(u') \quad \text{and} \quad k(u, |k(u)| - c_0) = k(u', |k(u)| - c_0) \quad . \quad (5.5)$$

Because s is bounded we have $C(u') \leq C(r) + O(1)$. For sufficiently large u we get

$$C(u') < \lg(\lg(|u|)) \quad . \quad (5.6)$$

From the formulas (5.4) and (5.5)

$$\forall i \geq 0. \quad \text{pr}_{12}(\delta_A^i((v, q(u), k(u')))) = \text{pr}_{12}(\delta_A^i((v, q(u), k(u))))$$

follows. Now (i) implies that v can be constructed when u' is known: We let A work on input u' until $q(u') = q(u)$ and $k(u') = k(u)$. Then we use (i) and reconstruct v with c many additional symbols. Putting it all together we get

$$\begin{aligned} C(v) &\leq C(u') + O(1) \\ &\leq \lg(\lg(|u|)) + O(1) \quad , \end{aligned}$$

where in the last step inequality (5.6) is used. This contradicts (iii). \square

Since the KC-DCF Lemma itself is difficult to apply, we also give the corresponding corrected formulation of its corollary (see [LiV95]): We say that a program p **decides** L_1 **for** L_2 , if on input $u \in L_2$ the program p terminates and decides $u \in L_1$ correctly.

Corollary 31 *Let $L \subseteq X^*$ be a deterministic context-free language, $x, y \in X^*$, and let c be a constant. Then there exists a constant c' with the following property: If $u, v, w \in X^*$, where u is suffix of $\dots yx$, then the following three conditions imply $C(w) \leq c'$:*

(i) $C(v''|p_{uv'}) \leq c$ for all decompositions $v'v'' = v$ and for all programs $p_{uv'}$ that decide $(uv')^{-1}L$ for prefix $\{v''\}$;

(ii) $C(w|p_{uv}) \leq c$ for all programs p_{uv} that list $(uv)^{-1}L$ in lexicographic order;

(iii) $C(v) \geq 2 \cdot \lg(\lg(|u|))$

PROOF: Let A be a dpda that recognizes L . If $L = \emptyset$ or $L = X^*$ nothing has to be shown. Otherwise the KC-DCF lemma 30 holds and we can show that under the given assumptions, (i) and (ii) from the corollary imply (i) and (ii) from the KC-DCF Lemma (proposition 30):

Asssume that, under the given premises, u, v satisfy (i) from the corollary. Let $v'v'' = v$, and let k, q be the required additional information from condition (i) from the KC-DCF lemma. Now, if by using only finitely many additional symbols we can construct a program $p_{uv'}$ that decides $(uv')^{-1}L$ for prefix $\{v''\}$, then the bound in condition (i) from the corollary induces the bound in condition (i) from the KC-DCF lemma. This construction of $p_{uv'}$ can be obtained by observing the dpda A on input v'' when starting with state q and stack k .

With a similar argument, (ii) from above implies condition (ii) from the KC-DCF lemma. \square

Though the original KC-DCF lemma and its corollary have been weakened in order to make them correct, they still can be applied for several nondeterministic context-free languages without much additional effort. We give two examples taken from [LiVi95]:

Example 3 *The set of palindroms $L = \{x \in X^* : x = x^{-1}\}$ is not deterministic context-free.*

PROOF: Set $y = 0$, $x = 1$, $u = 0^n 1$, and $v = 0^n$. For an incompressible n we have

$$C(v) + O(1) = C(\underline{n}) \geq \lg(n) = \lg(|u| - 1),$$

implying (iii) of the corollary for sufficiently large n . Let $v'v'' = v$, then v'' is the lexicographically first string in $(uv')^{-1}L$. Since all v are all 0, (i) is also satisfied. The lexicographically first string in $(uv)^{-1}L$ starting with 1 is 10^n , hence $w = 10^n$ satisfies (ii). On the other hand

$$C(w) + O(1) = C(\underline{n}) \geq \lg(n),$$

hence L can not be deterministic context-free. \square

Example 4 *$L = \{xy \in \{0, 1\}^* : |x| = |y|, y \text{ contains at least one } 1\}$ is not deterministic context-free.*

PROOF: Set $y = 0$, $x = 1$, $u = 0^n 1$, with $|u|$ is even, and $v = 0^{n+1}$. For incompressible n we have

$$C(v) + O(1) = C(\underline{n}) \geq \lg(n) = \lg(|u| - 1),$$

thus implying (iii) of the corollary for sufficiently large n . Let $v'v'' = v$. We only need one bit to code the information whether $|v'|$ is even or odd. If $|v'|$ is even (odd), then v'' is the shortest string of even (odd) length which is all 0 and not belongs to $(uv')^{-1}L$. This shows that also (i) is satisfied. The lexicographically first string which does not belong to $(uv)^{-1}L$ and starts with 1 is 10^{2n+3} . So with $w = 10^{2n+3}$, condition (ii) is satisfied, too. On the other hand we have

$$C(w) + O(1) = C(\underline{n}) \geq \lg(n),$$

hence L cannot be deterministic context-free. \square

5.4 Related Work

We do not know whether the connection between permutations and the context-free membership problem has been pointed out in a similar way before, other NP-complete problems related to formal languages are listed in [GaJo 79]. It seems interesting to study the permutation operator further by exploring other language classes or by imposing other restrictions on the functions f and g .

The fact that associativity of the CYK-product and regular languages are related is implicit in many discussions of parsing algorithms, see for instance the section on Earley parsing in [Har 78]. However, we do not know any previous generalization to idempotent semirings. For using semirings in order to model language restrictions, one might be interested in consulting the book [KuSa 85] by W. Kuich and A. Salomaa, which is an extensive monograph on semirings in connection with formal language theory. The KC-DCF lemma was first stated by M. Li and P. Vitányi, see [LiVi95]. A shortened proof of our version of the lemma can be found in [Gli 03].

Chapter 6

Valiant Parsing

In this chapter we turn to context-free semiring parsing. During the past decade, context-free semiring parsing has spawned new interest into parsing. Applications can be found in the fields of speech recognition, description of second order structures in computational biology, and for modelling recursive combinatorial objects. The **context-free semiring-parsing problem** is, for a given semiring-valued context-free grammar $P = (X, V, p, S)$ and a word $w \in X^*$, to compute $L(P)(w)$. For the definitions of semiring-valued grammars, see section 4.1. Throughout this chapter, we will assume that P is in Chomsky normalform. If the underlying semiring is the boolean semiring, then semiring parsing is equal to plain context-free word recognition.

Context-free parsing is complicated by the following fact: For arbitrary context-free grammars P , the CYK-product is — at least in general — not associative. Indeed, as we have seen in section 5.2, the class of languages for which associativity holds is very restricted. Hence semiring parsing requires to deal with non-associative multiplication. In [Val 74], Valiant has shown how transitive closure of triangular $n \times n$ matrices can be reduced to matrix-multiplication, even in the case that multiplication is non-associative. Valiant's reduction is such that it does not consume much more time than matrix multiplication, in the following sense: If matrix multiplication can be performed in time $O(n^{2+\varepsilon})$ for some $\varepsilon > 0$, then transitive closure is computable in time $O(n^{2+\varepsilon})$, too. Subsequently, Valiant showed in the same paper how plain context-free word recognition reduces to transitive matrix-closure, while maintaining the time bound $O(n^{2+\varepsilon})$.

Valiant did not address semiring parsing, probably because it was not a central focus of research back then. However, as we will see in section 6.1 of this chapter, his algorithm for transitive matrix closure can be applied for context-free semiring parsing without any modification. Thus it opens the possibility of context-free parsing in subcubic time for all semirings for which matrix multiplication can be performed in subcubic time — either through exploiting hardware parallelism or, if applicable, by algebraic techniques.

On the downside, Valiant's computation of transitive matrix closures via matrix multiplication involves a complicated recursion which is difficult to keep track of, especially since one is not used to deal

with the non-associative multiplication. In section 6.2 we give an alternative formulation of Valiant's reduction, which we hope is a more visual concept of the entire process. We build upon Rytter's paper on context-free parsing via computing shortest paths on grids (see [Ryt 95]). Again, Rytter's algorithm does not consider semiring parsing. Adapting Rytter's work to it is actually more complicated than adapting Valiant's transitive matrix closures, since the latter can be applied directly. We do this by using endomorphisms on (additively written) monoids in order to define path values. Like non-associative multiplication, endomorphisms distribute over sums, and allow to differentiate between right multiplication and left multiplication. This way we maintain enough information of the original problem in order to show that Rytter's approach is by no means weaker than Valiant's, in the sense that it can itself be applied for computing transitive matrix closures. We hope that our treatment of Valiant's reduction will help programmers to implement subcubic semiring parsing in a modular way.

In section 6.3 we give some applications of Valiant's reduction for context-free semiring parsing. For some applications, the so-called bottom-up property of the resulting parser is crucial. What exactly we mean by the term "bottom-up" is made more precise in section 6.2.

6.1 Semiring Parsing and Transitive Matrix Closures

Let $M = (M, +, 0)$ be a commutative monoid. Additionally, a binary operation $\times : M \times M \rightarrow M$ is defined, which distributes over $+$:

- (i) $(a + b) \times c = (a \times c) + (b \times c)$,
- (ii) $c \times (a + b) = (c \times a) + (c \times b)$, and
- (iii) $0 \times c = c \times 0 = c$

hold for all $a, b, c \in M$. It is not required that \times is associative. Still, \times is called a (non-associative) multiplication. Homomorphisms on M are only required to respect addition $+$ and may ignore \times .

First, we will introduce transitive closures of matrices over M . Then we proceed by showing how transitive closures of strictly upper triangular matrices can be used for constructing a table parser for context-free semiring parsing. Table parsing, briefly described, means to compute $L_A^P(u)$ for a given grammar $P = (X, V, p, S)$ and for each $A \in V$ and each infix u of a given input word w of length n . As a result of our reduction, an upper time bound of $O(2^{n+\varepsilon})$ with $\varepsilon > 0$ for $n \times n$ matrix multiplication over the underlying semiring also holds for semiring parsing for the given grammar P .

Let $(d_{ij}), (e_{ij})$ two $n \times n$ matrices with indices $1 \leq i, j \leq n$ and with entries in M . Their product is defined as usually as

$$(d \times e)_{ij} := \sum_{1 \leq k \leq n} d_{ik} \times e_{kj}.$$

We define the **transitive closure** of a matrix d by

$$d^+ := \sum_{i \geq 1} d^{(i)} .$$

Here, $d^{(i)}$ denotes the sum of all i th powers of d by considering each admissible bracketing of the product exactly once. More formally we define $d^{(1)} := d$ and recursively for $i > 1$:

$$d^{(i)} := \sum_{1 \leq j < i} d^{(j)} \times d^{(i-j)} \quad (6.1)$$

We then have

$$d^+ \times d^+ = \left(\sum_{i \geq 1} d^{(i)} \right) \times \left(\sum_{i \geq 1} d^{(i)} \right) = \sum_{k \geq 2} \sum_{i+j=k} d^{(i)} \times d^{(j)} = \sum_{k \geq 2} d^{(k)}$$

and thus $d^+ = d + d^+ \times d^+$ follows. Since the sum 6.1 is infinite, transitive closures do not necessarily exist for arbitrary matrices. However, for context-free table semiring-parsing we only need to deal with **strictly upper triangular matrices**. If d is a strictly upper triangular matrix (i.e. $d_{ij} = 0$ for $i \leq j$), then we have $d^{(n+1)} = 0$ and the sums can be computed as:

$$d^+ = \sum_{i=1}^n d^{(i)}$$

This means that each entry $d_{ij}^+ = d_{ij} + \sum_{i < k < j} d_{ik}^+ \times d_{kj}^+$ of d^+ is indeed well-defined.

Proposition 32 (Valiant) *Let (d_{ij}) be a strictly upper triangular matrix with entries in M . Assume $\varepsilon > 0$ exists so that matrix multiplication over M needs $O(n^{2+\varepsilon})$ time. Then the transitive closure d^+ can be computed in $O(n^{2+\varepsilon})$ time.*

We will proof proposition 32 in section 6.2 within the context of Rytter's shortest path algorithm for strongly congruent grids. We turn now to the semiring-parsing problem and show how to reduce semiring parsing to semiring matrix multiplication:

In the rest of this section, let $R = (R, +, \cdot, 0, 1)$ be an arbitrary semiring which we want to apply to semiring parsing for a given R -valued context-free grammar $P = (X, V, p, S)$. On the space $M = R^V$ of maps from nonterminals to semiring-values we define two binary operations $+, \times : M \times M \rightarrow M$. Addition $+$ will be componentwise, while \times is the CYK-product on $M = R^V$:

$$(i) \quad (c + d)(A) := c(A) + d(A)$$

$$(ii) \quad (c \times d)(A) := \sum_{B, C \in V} p(A \rightarrow BC) \cdot c(B) \cdot d(C) .$$

One can verify that $(M, +, 0)$, where 0 denotes the map which is always zero, is a commutative monoid and that \times distributes over $+$. Hence, according to Valiant's theorem 32, transitive closures of strictly upper triangular matrices over $M = R^V$ can be reduced to matrix multiplication over R^V .

The results above are fully sufficient for the reduction of context-free semiring parsing to semiring matrix multiplication: First, matrix multiplication over $M = R^V$ reduces to $O(|V|^3)$ matrix multiplications over R :

Let $(c_{ij}), (d_{ij})$ be two $n \times n$ matrices with entries $c_{ij}, d_{ij} : V \rightarrow R$. The entries of their product are given by

$$\begin{aligned} (c \times d)_{ij}(A) &:= \left(\sum_{1 \leq k \leq n} c_{ik} \times d_{kj} \right)(A) \\ &= \left(\sum_{1 \leq k \leq n} \sum_{B, C \in V} p(A \rightarrow BC) \cdot c_{ik}(B) \cdot d_{kj}(C) \right) \\ &= \sum_{B, C \in V} p(A \rightarrow BC) \cdot \left(\sum_{1 \leq k \leq n} c_{ik}(B) \cdot d_{kj}(C) \right). \end{aligned}$$

If we measure the size $|P|$ of the grammar P as the number of productions with non-zero value, i.e.

$$|P| := |\{r \in V \times (X \cup V \times V) : p(r) \neq 0\}|,$$

then from the equation above it follows that this reduction step involves not more than a factor of $|P| \in O(|V|^3)$.

In a final step we show that the transitive closure of an upper triangular matrix really solves the semiring parsing problem:

Let $w = x_1 \dots x_n \in X^*$. In order to compute $L(P)(w)$ we define a $(n+1) \times (n+1)$ matrix (c_{ij}) with entries $c_{ij} : V \rightarrow R$:

$$c_{ij}(A) := \begin{cases} p(A \rightarrow x_i), & \text{if } j = i + 1 \\ 0, & \text{else} \end{cases}$$

The following proposition shows that we obtain $L(P)(w)$ from the transitive closure c^+ :

Proposition 33 *For the matrix c defined as above and $1 \leq k, i, j \leq n+1$ we have*

$$c_{i,j}^{(k)}(A) = \begin{cases} L_A^P(x_i \dots x_{j-1}), & \text{for } j = i + k \\ 0, & \text{else} \end{cases} \quad (6.2)$$

In particular it follows $c_{i,i+1}^+(A) = L_A^P(x_i \dots x_j)$ for $1 \leq i \leq j \leq n$ and thus $L(P)(w) = c_{1,n+1}^+(A)$.

PROOF: The proof is by induction on k :

For $k = 1$ we get $L_A^P(x_i) = p(A \rightarrow x_i)$ directly from the definition of the matrix $c = c^{(1)}$.

For the inductive step we assume equation 6.2 holds for all $k' \leq k$. For $k+1$ it follows:

$$\begin{aligned} c_{ij}^{(k+1)}(A) &= \sum_{m=1}^k (c^m \times c^{(k+1-m)})_{ij}(A) \\ &= \sum_{m=1}^k \sum_{r=1}^{n+1} \sum_{B, C \in V} p(A \rightarrow BC) \cdot c_{ir}^{(m)}(B) \cdot c_{rj}^{(k+1-m)}(C) \end{aligned}$$

$$\begin{aligned}
 & \text{(by the inductive hypotheses: } c_{ir}^{(m)}(B) = \begin{cases} 0, & \text{if } r \neq i+m \\ L_B^P(x_i \dots x_{i+m-1}), & \text{else} \end{cases}) \\
 &= \sum_{m=1}^k \sum_{B, C \in V} p(A \rightarrow BC) \cdot L_B^P(x_i \dots x_{i+m-1}) \cdot c_{i+m,j}^{(k+1-m)}(C)
 \end{aligned}$$

Now, by the inductive hypothesis for $c_{i+m,j}^{(k+1-m)}(C)$, i.e.

$$c_{i+m,j}^{(k+1-m)}(C) = \begin{cases} L_C^P(x_{i+m} \dots x_{j-1}), & \text{if } j = i+k+1 \\ 0, & \text{else} \end{cases}$$

it follows $c_{ij}^{(k+1)} = 0$ for $j \neq i+k+1$ while for $j = i+k+1$ we get

$$\begin{aligned}
 c_{ij}^{(k+1)}(A) &= \sum_{m=1}^k \sum_{B, C \in V} p(A \rightarrow BC) \cdot L_B^P(x_i \dots x_{i+m-1}) \cdot L_C^P(x_{i+m} \dots x_{j-1}) \\
 &= \sum_{m=i}^{j-1} \sum_{B, C \in V} p(A \rightarrow BC) \cdot L_B^P(x_i \dots x_{m-1}) \cdot L_C^P(x_m \dots x_{j-1}) \\
 &= L_A^P(x_i \dots x_{j-1})
 \end{aligned}$$

since P is in Chomsky normalform. □

By applying the reductions above, we get the following bounds for semiring-parsing:

Proposition 34 *Let R be a semiring for which $n \times n$ matrix multiplication needs $O(n^{2+\varepsilon})$ time for some $\varepsilon > 0$. Let $P = (X, V, p, S)$ be a R -valued context-free grammar in Chomsky normalform and $w \in X^n$. Then $L(P)(w)$ can be computed in $O(n^{2+\varepsilon}|P|)$ time. □*

6.2 Transitive Closure via Shortest Paths

6.2.1 Bottom-Up Properties

The results of the previous section together with Valiant's reduction for transitive closure ensure that we can, for all semirings, reduce the semiring parsing problem to semiring matrix multiplication. There are two reasons why we want to re-examine Valiant's reduction in the following section: The first one is that Valiant's original description of the reduction algorithm is not very visual to the reader. In the past, other researchers have reformulated Valiant's algorithm, see section 6.4. However, none of the authors, including Valiant, considered semiring parsing. In the context of the emerging interest in parsing methods for semiring parsing, the question rose naturally how to investigate how well the alternative approaches to Valiant's reduction generalize. In this section we will show that Rytter's approach cannot only be generalized for context-free semiring parsing, but also for the more general problem of computing transitive closures of upper triangular matrices.

The second reason for re-examination of Valiant's reduction is that algebraic methods in computer science often hide computationally relevant information. We give an example:

Sometimes a data type forms a semiring, but not for all instances of the data type the semiring operations can be performed efficiently. A good example are syntax trees $T \in \mathcal{T}(X \cup V, V)$ which may contain holes (nonterminals as leaves). Two sequences $(T_1, \dots, T_m), (T'_1, \dots, T'_n) \in (\mathcal{T}(X \cup V, V))^*$ can be multiplied by plugging, from left to right, the syntax trees of the right sequence into the holes of the left sequence, and then forming a new sequence out of the newly constructed trees and of what remains from the right sequence. Together with the empty sequence, this multiplication forms a monoid on which we can define suitable semirings, for instance the semiring of sets of syntax-trees. It is clear that such a multiplication of sequences of trees is destructive for the trees of the left sequence and hence all the syntax trees need to be traversed, which can become time-consuming. However, this situation never occurs during bottom-up parsing, where trees are build from ground up. Here the only trees with holes are actually the grammar's productions: We consider the inner multiplication of the CYK-product as it appears in CYK-bottom-up parsing,

$$p(A \rightarrow BC) \cdot L_B^P(u) \cdot L_C^P(v), \quad \text{for appropriate } u, v \in X^+$$

where computation is done within some underlying semiring. In this case we observe:

1. Each value $L_B^P(u), L_C^P(v)$ corresponds to a single syntax-tree without any holes.
2. The only syntax trees which contain holes are those of the form $A \rightarrow BC$, i.e. they contain only the two holes B and C .

We conclude that for CYK-parsing, destructive operations on syntax trees only happen within a constant upper limit in size. However, this desirable bottom-up property is hidden by being embedded in an algebraic framework. So a programmer who only uses the algebraic framework as the parser's interface might choose the wrong implementation for his underlying semiring. Also it is impossible to derive tight upper time bounds if the necessary implementation details are hidden.

Valiant's reduction algorithm respects the bottom-up property of the CYK-product, in the sense that it does not construct semiring elements which correspond to large syntax trees with holes. The property has indeed an application, for instance if we want to use algorithms for fast boolean matrix multiplication and multiplication product witnesses for simultaneous construction of at least one syntax tree for each positive entry of the CYK parsing table. The bottom-up property is formulated below:

Definition 17 Let $M = (M, 0, +)$ be a commutative monoid with an additional distributive multiplication \times . Let \mathcal{A} be an algorithm for computing the transitive closure of an upper triangular matrix (c_{ij}) over M via matrix multiplication. Then \mathcal{A} is **M-bottom-up** if for each matrix multiplication which \mathcal{A} calls, all the entries of the operands are generated by the entries of the input matrix (c_{ij}) .

For a semiring R and a R -valued grammar $P = (X, V, p, S)$, a parsing algorithm is **RP-bottom-up** (or shortly **bottom-up**) if all intermediate values in R are the p -values of syntax-trees from $\mathcal{T}(X \cup V, V)$ with at most a constant number of nonterminals as leaves. The definitions above also apply to any subalgorithm which is involved in the computation.

The bottom-up property of a transitive-closure reduction which works for arbitrary M is very natural: Without any knowledge about M , where — if not from the input itself — should intermediate values be generated from?

The bottom-up property of a semiring-parser is more restrictive. Indeed there exist many parsing algorithms, like Early parsing, which are not bottom-up. However, the construction of $M = R^V$ and the initial matrix (c_{ij}) in the previous section ensure that each subsequent M -bottom-up Valiant reduction results in a bottom-up parser, provided that matrix multiplication is M -bottom-up. This observation follows directly from the definition of the CYK-product on $M = R^V$ and the discussion above.

6.2.2 Strongly Congruent Grids and Shortest Paths

We will now give a proof for Valiant's reduction in terms of a suitable algorithm. Valiant himself already gave a proof in his paper [Val 74], but we will use ideas from Rytter's parsing algorithm (see [Ryt 95]). Since Rytter formulated his algorithm for plain context-free parsing, we need a considerable amount of generalizations in order to show that Rytter's algorithm is also suitable to compute transitive closures of triangular matrices over arbitrary M . The advantage of Rytter's approach lies in Valiant's reduction being formulated in terms of a shortest path problem. The graphs which we consider will be two-dimensional grids. This is more intuitive than working with non-associative multiplication, which we will get rid of by replacing it by endomorphisms.

Let $(M, +, 0)$ be a commutative monoid and \times be a distributive multiplication. The distributive laws ensure that for each $a \in M$, left-multiplication $x \mapsto ax$ and right-multiplication $x \mapsto xa$ are endomorphisms (i.e. homomorphisms from M onto itself). For the shortest path problem on strongly congruent grids, we replace \times by two monoids of endomorphisms: We define H_1 and H_2 to be the sets of all left- and right-multiplications on M respectively. With \cdot as the concatenation of maps and 1 being the identity map, $(H_1, \cdot, 1)$ and $(H_2, \cdot, 1)$ are both monoids of M -endomorphisms. Given a set of M -endomorphisms H and two $n \times n$ matrices $(h_{ij}), (c_{ij})$ with $h_{ij} \in H$ and $c_{ij} \in M$, we define their matrix product as

$$(h \times c)_{ij} = \sum_{k=1}^n h_{ik}(c_{kj}) . \quad (6.3)$$

Definition 18 A $n \times n$ grid is a directed graph $G = (N, E)$ with nodes $N = \{1..n\} \times \{1..n\}$ and edges $E = E_1 \cup E_2$ with $E_1 = \{((i, j), (i, j+k)) : k > 0\}$ and $E_2 = \{((i+k, j), (i, j)) : k > 0\}$. For a commutative monoid $M = (M, +, 0)$ and two sets H_1, H_2 of M -endomorphisms, a strongly congruent grid is a grid $G = (N, E)$ with a node labelling $K : N \rightarrow M$ and an edge labelling $h : E \rightarrow H_1 \cup H_2$ such that:

- (i) $h(E_i) \subseteq H_i$ for $i = 1, 2$.
- (ii) $h((i, j), (i+k, j)) = h((i, j'), (i+k, j'))$ for all $1 \leq i, j, j', k \leq n$ with $i+k \leq n$.
- (iii) $h((i, j), (i, j+k)) = h((i', j), (i', j+k))$ for all $1 \leq i, i', j, k \leq n$ with $j+k \leq n$.

In other words, if $G = (N, E)$ is a strongly congruent grid, then to parallel edges the same label is assigned. Hence we can write for all i, k with $i + k \leq n$

$$h(i, \cdot, i + k, \cdot) := h((i, j), (i + k, j)) \quad h(\cdot, i, \cdot, i + k) := h((j, i), (j, i + k)) \quad (6.4)$$

independently from j . The map $h : E \rightarrow H_1 \cup H_2$ can be extended to all paths π in G : Let $\pi = \pi_1 \dots \pi_r$ be a path in G , represented as a sequence of edges $\pi_i \in E$. Then we set

$$h(\pi) := h(\pi_1) \cdots h(\pi_r) .$$

This way we get a map $h : E^* \rightarrow (H_1 \cup H_2)^*$, where $(H_1 \cup H_2)^*$ denotes the monoid of all endomorphisms generated by $H_1 \cup H_2$. So each path π in G describes an endomorphism $h(\pi) : M \rightarrow M$, denoted by h_π . For a path π of length 0, h_π is simply the identity map $\text{id} : x \mapsto x$.

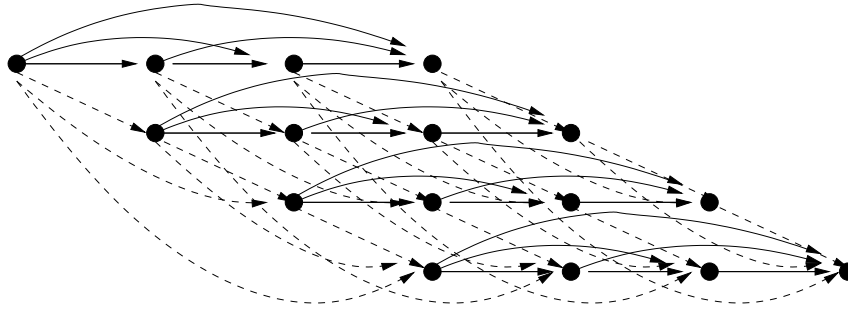


Figure 6.1: A 4×4 grid with all edges depicted. For a strongly congruent grid, parallel edges have the same labels.

Definition 19 *The shortest-path problem for strongly congruent grids is described as follows: Let $M = (M, +, 0)$ be a commutative monoid and H_1, H_2 be sets of endomorphisms on M . Let $G = (N, E)$ be a strongly congruent grid with node labels $K : N \rightarrow M$ and edge labels $h : E \rightarrow H$. The problem is to compute for all $p \in N$ the value*

$$C(p) := \sum_{s \in N} \sum_{\pi: s \rightarrow p} h_\pi(K(s)) .$$

Note that a grid does not contain any loops, hence we can also write

$$C((i, j)) := \sum_{s_1 \leq i} \sum_{s_2 \leq j} \sum_{\pi: (s_1, s_2) \rightarrow (i, j)} h_\pi(K((s_1, s_2)))$$

for all nodes $(i, j) \in N$. Let G be a $n \times n$ grid $G = (N, E)$ with edge labelling $h : E \rightarrow H_1 \cup H_2$ and node labelling $K : N \rightarrow M$ and let C be its corresponding solution to the shortest-path problem. Let $x \leq x', y \leq y' \in \{1..n\}$ describe a range of indices $\{(i, j) : x \leq i \leq x' \wedge y \leq j \leq y'\}$. We assume that for all i, j with $x \leq i \leq x'$ and $y \leq j \leq y'$ the following values are known:

$$K'((i, j)) := K(i, j) + \sum_{s_1 < x \vee s_2 < y} h_{((s_1, s_2), (i, j))}(C((s_1, s_2))) \quad (6.5)$$

Note that we do not use the $C(i, j)$ -values in the range $x \leq i \leq x'$ and $y \leq j \leq y'$. Instead, we compute the shortest-path values $C'(i, j)$ of the subgrid induced by the nodes (i, j) with $x \leq i \leq x'$ and $y \leq j \leq y'$, i.e.

$$C'((i, j)) := \sum_{\substack{x \leq s_1 \leq i \\ y \leq s_2 \leq j}} \sum_{\pi: (s_1, s_2) \rightarrow (i, j)} h_{\pi}(K'((s_1, s_2))) .$$

We find that for $x \leq i \leq x'$ and $y \leq j \leq y'$, the values $C'((i, j))$ and $C((i, j))$ are equal:

$$\begin{aligned} C'((i, j)) &= \sum_{\substack{x \leq s_1 \leq i \\ y \leq s_2 \leq j}} \sum_{\pi: (s_1, s_2) \rightarrow (i, j)} h_{\pi} \left(K(s_1, s_2) + \sum_{s'_1 < x \vee s'_2 < y} h_{((s'_1, s'_2), (s_1, s_2))}(C((s'_1, s'_2))) \right) \\ &= \sum_{\substack{x \leq s_1 \leq i \\ y \leq s_2 \leq j}} \sum_{\pi: (s_1, s_2) \rightarrow (i, j)} h_{\pi}(K(s_1, s_2)) \\ &\quad + \sum_{\substack{x \leq s_1 \leq i \\ y \leq s_2 \leq j}} \sum_{\pi: (s_1, s_2) \rightarrow (i, j)} h_{\pi} \left(\sum_{s'_1 < x \vee s'_2 < y} h_{((s'_1, s'_2), (s_1, s_2))} \left(\sum_{\substack{s''_1 \leq s'_1 \\ s''_2 \leq s'_2}} \sum_{\pi': (s''_1, s''_2) \rightarrow (s_1, s_2)} h_{\pi'}(K(s''_1, s''_2)) \right) \right) \\ &= \sum_{\substack{x \leq s_1 \leq i \\ y \leq s_2 \leq j}} \sum_{\pi: (s_1, s_2) \rightarrow (i, j)} h_{\pi}(K(s_1, s_2)) \\ &\quad + \sum_{s'_1 < x \vee s'_2 < y} \sum_{s'_1 < x \vee s'_2 < y} \sum_{\substack{s''_1 \leq s'_1 \\ s''_2 \leq s'_2}} \sum_{\substack{\pi: (s_1, s_2) \rightarrow (i, j) \\ \pi': (s''_1, s''_2) \rightarrow (s'_1, s'_2)}} h_{\pi' \cdot ((s'_1, s'_2), (s_1, s_2)) \cdot \pi}(K(s''_1, s''_2)) \\ &= \sum_{\substack{x \leq s_1 \leq i \\ y \leq s_2 \leq j}} \sum_{\pi: (s_1, s_2) \rightarrow (i, j)} h_{\pi}(K(s_1, s_2)) + \sum_{s_1 < x \vee s_2 < y} \sum_{\pi: (s_1, s_2) \rightarrow (i, j)} h_{\pi}(K(s_1, s_2)) \\ &= \sum_{\substack{s_1 \leq i \\ s_2 \leq j}} \sum_{\pi: (s_1, s_2) \rightarrow (i, j)} h_{\pi}(K(s_1, s_2)) = C((i, j)) \end{aligned}$$

Assuming the shortest-path values $C((i, j))$ for $x \leq i \leq x'$ and $y \leq j \leq y'$ are the only missing ones, we can compute them from the values $K'((i, j))$ (equation 6.5) and the solution of the shortest path problem for the corresponding subgrid. This suggests a recursive splitting of a grid for solving the shortest path problem. All we have to do is to maintain equation 6.5 for the subgrid in question. We will see below how this can be done by matrix multiplications of the form as defined in equation 6.3. For a better understanding, please note that equation 6.3 defines an application of a matrix of endomorphisms to a matrix of monoid values.

Proposition 35 (Generalized Rytter Reduction) *Let G be a $n \times n$ strongly congruent grid with node labels in a commutative monoid M and edge labels in H_1, H_2 , where H_1, H_2 are sets of M -endomorphisms. Then the shortest-path problem for G can be reduced to multiplication of matrices as defined in equation 6.3, where entries of the first operand matrix are either all from H_1 or all from H_2 , and the entries of the second operand matrix are all from M . For the reduction, the following time bound holds: If those matrix multiplications for $n \times n$ matrices can be computed in time $O(n^{2+\varepsilon})$ for some $\varepsilon > 0$, then the shortest-path problem for G can be computed in time $O(n^{2+\varepsilon})$. The reduction is M -bottom-up.*

PROOF: Let $G = (N, E)$ be a grid and M, H_1, H_2 be given as described in the precondition of the proposition. Let $K : N \rightarrow M$ and $h : E \rightarrow H_1 \cup H_2$ be labellings such that G, K, h is a strongly congruent grid and let $C : N \rightarrow M$ be its solution for the shortest-path problem. We want to describe an algorithm which computes all values $C(i, j)$, for $(i, j) \in N$ by using matrix-multiplication of the form of equation 6.3. For the sake of simplicity we assume that G is a $n \times n$ grid where n is a power of 2. This is not a loss of generality since we can always solve the shortest path problem by enlarging the grid until its size is a power of 2. The extra costs for solving the shortest-path problem on the larger grid are then negligible. For $n \leq 1$ we do not need to do anything. For all other $n \leq 2$ we proceed by recursion. We can assume that $G = (N, E)$ is a $2n \times 2n$ grid and that we can already solve the shortest-path problem for $n \times n$ grids by recursive calls to our algorithm. We define the following sets:

$$N_1 := \{(i, j) \in N : i \leq n \wedge j \leq n\}$$

$$N_2 := \{(i, j) \in N : i \leq n \wedge n < j \leq 2n\}$$

$$N_3 := \{(i, j) \in N : n < i \leq 2n \wedge j \leq n\}$$

$$N_4 := \{(i, j) \in N : n < i \leq 2n \wedge n < j \leq 2n\}$$

The idea is to compute, in the order listed here, the values $C(i, j)$: first for all $(i, j) \in N_1$, then for all $(i, j) \in N_2$, for all $(i, j) \in N_3$, and finally for all $(i, j) \in N_4$.

For $(i, j) \in N_1$, the values $C(i, j)$ coincide with the values of the shortest-path problem for the subgrid induced by N_1 . Hence we can compute them by recursively calling our algorithm for the smaller $n \times n$ subgrid.

For $(i, j) \in N_2$, we proceed as follows: First, for all such $(i, j) \in N_2$, we compute

$$K'((i, j)) := K((i, j)) + \sum_{k=1}^n h(., k, ., j)(C((i, k))) .$$

Note that $K'((i, j))$ can be computed for all $(i, j) \in N_2$ simultaneously by a matrix multiplication of the form in equation 6.3 and then adding the original values $K((i, j))$. By setting $x = 1, x' = n, y = n + 1, y' = 2n$, K' satisfies equation 6.5. We can now obtain the values $C((i, j))$ for $(i, j) \in N_2$ by recursively computing the shortest-path values for the subgrid induced by N_2 together with the new labels K' .

For $(i, j) \in N_3$, we proceed in a similar way: For all such $(i, j) \in N_3$, we compute

$$K'((i, j)) := K((i, j)) + \sum_{k=1}^n h(k, ., i, .)(C((k, j)))$$

by using a single matrix multiplication. Again, we obtain the values $C((i, j))$ for $(i, j) \in N_3$ by recursively computing the shortest-path values for the subgrid induced by N_3 and node labels K' .

Finally, for $(i, j) \in N_4$ we compute

$$K'((i, j)) := K((i, j)) + \sum_{k=1}^n h(., k, ., j)(C((i, k))) + \sum_{k=1}^n h(k, ., i, .)(C((k, j)))$$

by using two matrix multiplications. A last recursive call of the algorithm will then compute the missing values $C((i, j))$ for $(i, j) \in N_4$.

In each recursive step with a $n \times n$ grid as its input, the algorithm above requires exactly $4 \cdot n/2 \times n/2$ matrix multiplications of the form of equation 6.3. According to our assumptions, this takes $O(n^{2+\epsilon})$ time for some $\epsilon > 0$. Also in each recursive step, the algorithm calls itself four times, each time for a subgrid of size $n/2 \times n/2$. The running time $T(n)$ is therefore bounded by

$$T(n) = 4 \cdot O((n/2)^{2+\epsilon}) + 4T(n/2)$$

while $T(n) = O(1)$ for $n = 1$. This yields $T(n) = O(n^{2+\epsilon})$. Finally we conclude that the algorithm is M -bottom-up by observing that all intermediate values in M are generated by the values of the algorithm's input grid. \square

Rytter formulated the boolean case of the shortest path problem for strongly congruent grids as a mean to reduce plain context-free parsing to boolean matrix multiplication. Though the reduction is much more visible, it does not generalize that easily to semiring parsing. We show that with our generalizations, the more general transitive closure problem reduces to it. As discussed in section 6.1, this will in turn allow semiring parsing.

Proposition 36 *Let $M = (M, +, 0)$ be a commutative monoid and \times be a binary multiplication which distributes over $+$. Let H_1, H_2 be some sets of, respectively, left- and right-multiplications over M . If the shortest-path problem for strongly congruent grids G with node labels in M and edge labels in H_1, H_2 can be solved in $O(n^{2+\epsilon})$ time for some $\epsilon > 0$, then transitive matrix closures for upper triangular matrices over M can be computed in $O(n^{2+\epsilon})$ time. The reduction is M -bottom-up.*

Before we prove proposition 36, we note that now Valiant's theorem 32 follows directly from the propositions 35 and 36. Indeed, matrices of left-multiplications can be encoded as matrices over M , such that equation 6.3 refers to a simple semiring matrix multiplication. The same holds for application matrices over right-multiplications, with the only difference that the matrix of multipliers is multiplied to the right. We can also see that Valiant's reduction is M -bottom-up.

PROOF: (of proposition 36)

Let $M = (M, +, 0)$ be a commutative monoid and \times be a distributive multiplication and H_1, H_2 be as defined in the preconditions of the proposition. Let (d_{ij}) be a $n \times n$ strictly upper triangular matrix with entries in M . For the sake of simplicity, we assume that n is a power of 2. If not, we can append $O(n)$ zero rows and zero columns to d .

We show proposition 36 by induction. For $n \leq 1$, nothing has to be done in order to compute the transitive closure d^+ . In our inductive step, we assume the truth of proposition 36 for matrices of size

$n \times n$. So we continue with for $2n \times 2n$ matrices:

Let d be a $2n \times 2n$ matrix, $n \geq 2$, with entries in M . For indices i, j with $1 \leq i, j \leq n \vee n \leq i, j \leq 2n$ we can compute the correct values d_{ij}^+ recursively by computing the transitive closures of the corresponding submatrices. What is left to compute are the missing values d_{ij}^+ for $1 \leq i < n \wedge n < j \leq 2n$: For the $n \times n$ grid $G = (N, E)$ we define the following node labels $K : N \rightarrow M$ and edge labels $h : E \rightarrow H_1 \cup H_2$ by setting

$$K((i, j)) := d_{n+1-i, n+j} \quad \text{for } 1 \leq i, j \leq n,$$

and

$$h(k, \cdot, i, \cdot) := x \mapsto d_{n+1-i, n+1-k}^+ \times x \quad \text{and} \quad h(k, \cdot, j, \cdot) := x \mapsto x \times d_{n+k, n+j}^+$$

for $1 \leq k < i, j \leq n$, see equation 6.4. Our claim is now that the solution

$$C(p) := \sum_{s \in N} \sum_{\pi: s \rightarrow p} h_\pi(K(s))$$

of the shortest-path problem for the strongly congruent grid G with labels h, K yields the missing entries of d^+ : It is

$$C(p) = d_{n+1-i, n+j}^+$$

for all nodes $p = (i, j)$ with $1 \leq i, j \leq n$. We show this by induction on the difference $j - i$.

$j = i = 1$: Since in the $n \times n$ grid G there is only the path of length 0 from the node $(1, 1)$ to itself, we get

$$C((i, j)) = \text{id}(K((1, 1))) = d_{n, n+1} = d_{n, n+1}^+.$$

For the inductive step, we assume $C((i, j)) = d_{n+1-i, n+j}^+$ for $2 \leq i + j < r$. We then conclude for $i + j = r$:

$$\begin{aligned} C((i, j)) &= K((i, j)) + \sum_{k < i} h(k, \cdot, i, \cdot) C((k, j)) + \sum_{k < j} h(\cdot, k, \cdot, j) C((i, k)) \\ &= d_{n+1-i, n+j} + \sum_{k < i} d_{n+1-i, n+1-k}^+ \times d_{n+1-k, n+j}^+ \\ &\quad + \sum_{k < j} d_{n+1-i, n+k}^+ \times d_{n+k, n+j}^+ \\ &= d_{n+1-i, n+j} + \sum_{n+1-i < k < n+j} d_{n+1-i, k}^+ \times d_{k, n+j}^+ \\ &= d_{n+1-i, n+j}^+ \end{aligned}$$

The algorithm is M -bottom-up. This can be verified by inspection, i.e. one can see values from M are never used which are not generated by the input values. Finally, after establishing correctness of the algorithm, we derive the required time bounds: For a given $n \times n$ input matrix, $n = 2^k$, in each recursive step the algorithm calls itself twice for matrices of the size $n/2 \times n/2$, followed by a call to the shortest-path algorithm for a grid of size $n/2 \times n/2$. The time for constructing the grid G is negligible. Under the assumption of the proposition our algorithm has an upper timebound $T(n)$ which satisfies $T(n) = 2T(n/2) + O(n^{2+\varepsilon})$ for all $n = 2^k$ and $T(1) = O(1)$. It follows that T itself satisfies $T(n) = O(n^{2+\varepsilon})$. \square

6.3 Applications

Only briefly we will mention some applications of Valiant parsing for some semirings for which matrix multiplication can be computed in subcubic time. For a thorough discussion of semiring parsing, the reader is referred to [Goo 98].

For the boolean semiring $\mathbb{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$ as the underlying semiring we are down to plain context-free parsing for context-free grammars $P = (X, V, P, S)$. Boolean matrix-multiplication can be embedded into matrix multiplication over the integers, and known subcubic matrix multiplication algorithms involve only a modest growth of the size of numbers during computation. Hence Valiant parsing can be used in order to construct a recognition matrix for an input string $w \in X^*$ in subcubic time. The recognition matrix represents in a certain way all possible parses, but an arbitrary syntax tree (if it exists) can be constructed in $O(n)$ time.

In some applications it might be desirable to have one — or a finite number of — valid syntax trees for each infix of the input string w . This is useful if the input is subject to change or for error analysis in language processing. It turns out that Valiant parsing can simultaneously construct an arbitrary syntax-tree for each infix of w and nonterminal $A \in V$. Our discussion tries to avoid technical details, which will actually involve to re-examine the entire algorithm again:

We start with the monoid $\mathcal{T}^* = (\mathcal{T}(X \cup V, V)^*, \cdot, \varepsilon)$ of syntax trees which are allowed to contain holes (nonterminals as leaves). As mentioned in section 6.2, for two sequences $(T_1, \dots, T_m), (T'_1, \dots, T'_n) \in \mathcal{T}^*$, their product is defined by plugging the trees $T'_1, \dots, T'_{\min(n', r)}$ from left to right into the r holes of T_1, \dots, T_m . Their product is the resulting sequence concatenated with the remaining sequence $(T'_{\min(n', r)+1}, \dots, T'_n)$. One can verify that \mathcal{T}^* is indeed a monoid. Based on \mathcal{T}^* we can define the semiring $(2^{\mathcal{T}^*}, \cup, \cdot, \emptyset, \cdot)$, where for two sets $F, G \subseteq \mathcal{T}^*$ we define

$$F \cdot G := \{ (T_1, \dots, T_m) \cdot (T'_1, \dots, T'_n) : (T_1, \dots, T_m) \in F, (T'_1, \dots, T'_n) \in G \}.$$

Using $2^{\mathcal{T}^*}$ as the underlying semiring, Valiant's algorithm will compute the set of *all* syntax trees for each $A \in V$ and each infix of the input word w by using matrix multiplication over $2^{\mathcal{T}^*}$. This, however, cannot be implemented in polynomial time since the number of possible syntax trees may grow exponentially.¹ If we are satisfied by obtaining only one syntax tree (if it exists) as a suitable representative for the entire class of valid syntax tree from $A \in V$ to each infix u , we could naively identify all syntax trees with equal root and equal front. The resulting structure will still be a semiring. However, we must be a bit more careful since the actual form of the representative is not a property of the underlying semiring. The parsing algorithm might well mess up and yield a representative where root and front fit, but is not valid for the grammar P . In fact, we need to inspect the parsing algorithm in order to convince ourselves that it does not mess up and produces invalid syntax trees. We omit the proof here. Another solution might be to either construct a more complicated semiring (which we doubt being possible), or

¹At least not for “natural” representations of syntax trees. But in some way, the recognition matrix itself can be understood as a proper representation of all possible syntax trees.

to find a better interface than the semiring framework.

Given two $n \times n$ matrices $(c_{ij}), (d_{ij})$ which entries are either a syntax tree or zero, it remains to show that they can be multiplied in subcubic time. The task is to compute a product matrix (e_{ij}) which has a zero entry at e_{ij} if and only if no $k = 1..n$ with $c_{ik} \neq 0$ and $d_{kj} \neq 0$ exists. Otherwise one is allowed to choose some k with $c_{ik} \neq 0$ and $d_{kj} \neq 0$ and combine the two syntax-trees c_{ik} and d_{kj} in order to form e_{ij} . This can be achieved by using boolean matrix multiplication and the subsequent computation of boolean matrix multiplication witnesses. For both, subcubic algorithms exists since the boolean-matrix-multiplication-witness problem actually reduces to boolean matrix multiplication (a probabilistic algorithm can be found in [MoRa 95]). Also, since Valiant's reduction yields a bottom-up parser, newly constructed syntax-trees can be build non-destructively.

Finally we mention the following semirings which allow subcubic parsing:

1. The semiring of natural numbers with addition and multiplication as usual. This allows to count the number of syntax trees for a given input word w and for all its infixes and nonterminals $A \in V$, simultaneously. Note that the numbers can grow exponential and hence the number of bits for their representation grows lineary. If we do not need to obtain exact results for very large numbers, we can either bound the maximum representable number, or work with approximations like floating point numbers.
2. The semiring $([0..1], +, \cdot, 0, 1)$ which can represent probabilities of derivations.

6.4 Related Work

Valiant's reduction involves a complicated recursion scheme. There have been many subsequent reformulations of it in order to make his result more accessible. See for instance the book [Har 78] and the paper [Wal 84]. The idea of describing Valiant's reduction in terms of a shortest-path problem is from [Ryt 95]. Semiring parsing in general is treated in depth in the thesis [Goo 98]. The paper [Lee 02] shows that plain context-free parsing (under very reasonable assumptions²) is at least as difficult as boolean matrix multiplication. It might be interesting to investigate how far the latter result extends to general semiring parsing.

²Lee assumes that grammar size induces at most a linear factor in parsing time and that the parser produces information about the parsing process.

Appendix A

Definitions from Graph Theory

We recall, mostly in order to settle notation, some basic definitions of graph theory:

Definition 20 (directed and undirected graphs) A **directed graph** (or: *digraph*) $G = (N, E)$ consists of a finite set of **nodes** N and a set of **edges** $E \subseteq N \times N$ of ordered node-pairs. An **undirected graph** $G = (N, E)$ has a finite set of nodes N and the set of edges is a set $E \subseteq \{\{p, q\} : p \neq q \in N\}$ of unordered node-pairs. In the undirected case we sometimes write (p, q) for some edge $\{p, q\} \in E$ so that we identify (p, q) and (q, p) . For $(p, q) \in E$ we then say that the nodes p, q are **neighbours** and that they are **joined** by the edge $e = (p, q)$. An edge of the form $e = (p, p)$ is called a loop.

Note that according to our definition, undirected graphs never contain any loops. This might be a defect if we want to go from a digraph to the underlying undirected graph by forgetting the directions of the edges. However, we never use undirected graphs with loops in this thesis and by this way the following definitions work conveniently for both, directed and undirected graphs:

Definition 21 (paths and cycles) Let $G = (N, E)$ be a graph (directed or undirected). A graph $G' = (N', E')$ with $N' \subseteq N$ and $E' \subseteq E$ is called a **subgraph** of G . Furthermore, G' is an **induced subgraph** if $E' = \{(p, q) \in E : p, q \in N'\}$. A **path** in G from $p \in N$ to $q \in N$ is a sequence of nodes $\pi = \pi_0, \dots, \pi_k \in N^*$ such that $(\pi_{i-1}, \pi_i) \in E$ for all $i = 1..k$, and $\pi_0 = p, \pi_k = q$. The length of π is denoted by $|\pi| := k$. The fact that a path π starts in node p and ends in q is denoted by $\pi : p \rightarrow_G q$, where the subscript G is sometimes omitted in the case that the context is clear. An edge $e \in E$ is **traversed** by a path $\pi = \pi_0, \dots, \pi_k$ if $e = (\pi_{i-1}, \pi_i)$ for some $i = 1..k$. For a graph G the set of all paths contained in G is denoted by $\Pi(G)$. A path is **simple** if it traverses no edge more than once. G is **connected** when there always exists a path in G between arbitrary nodes $p, q \in N$. The path $\pi = \pi_0, \dots, \pi_k$ is **closed** when $\pi_0 = \pi_k$. Furthermore, a closed path $\pi = \pi_0, \dots, \pi_{k-1}, \pi_0$ in a directed graph G is a **cycle** if $p_i \neq p_j$ for all $0 \leq i < j < k$. In the undirected case we additionally demand that $|\pi| \neq 2$, thus a cycle can be identified (up to rotation of nodes) with a connected subgraph where each node has exactly two neighbors. A graph (directed or undirected) is **acyclic** if it contains no cycles.

Definition 22 (edge labeling) For a graph $G = (N, E)$ and a set R , an **edge labeling** with values in R is a map $X : E \rightarrow R$. In the case that $R = (R, \cdot, 1)$ is a monoid, we extend X to a mapping $X : \Pi(G) \rightarrow R$ by

$$X(\pi) := X(\pi_0, \pi_1) \cdot \dots \cdot X(\pi_{k-1}, \pi_k)$$

for all paths $\pi = \pi_0, \dots, \pi_k \in \Pi(G)$. Hence, for a path π the path-value of the labeling X is basically the labeling of the edges contained in π multiplied in their order of traversal. Note that if the path π has length 0, then $X(\pi) = 1$.

Appendix B

Deamortized Bothsided Extendible Arrays

While in this thesis we deal with persistent arrays, we are often interested in less versatile implementations of arrays. For instance, a (functional) programming environment may only offer persistent arrays to the programmer, but often not all of their features are needed by a particular routine. In those cases where this is detected by the interpreter, one can use alternative forms of static or dynamic arrays.

A particular interesting ADT is that of both-sided extendible arrays, since it can be used for the implementation of stacks, queues, and an intermediate layer for memory management of relocatable objects.

The simplest form of arranging n elements a_0, a_1, \dots, a_{n-1} , all of the same type (and size) of data, is to store them in an array of subsequent memory locations. Thus the i th element a_i can be accessed by computing its address in memory, which is the address of the first element a_0 plus the index i (times the size of each element). Because the arrangement of the n elements in the previously allocated memory remains fixed, an array is a **static** datastructure. It is clear that a fixed memory layout becomes an disadvantage if one wants to append new elements to the array: The naive way to extend an array is to copy the whole array to a larger area of memory every time we insert an element. In the context of this section, an array $A[0 \dots n-1]$ is an abstract datatype (ADT) containing n elements $A[0], A[1], \dots, A[n-1]$, **all of same type and size**. Given the index i , computing the address of $A[i]$'s memory location should be considerable fast, i.e. should only take few machine instructions or (in hardware) small depth circuits. Also iteration (i.e. subsequent access to all elements) should be fast and — if possible — take advantage of memory prefetching and caching.

Implementing Deques

Bothsided extendible arrays are called **deques**. A deque is an array $D[0 \dots n-1]$ equipped with the following **deque-operations**:

- **ADDATFRONT(e)**: Increase the size n of D by one. Then shift all elements one position to the right and set $D[0] := e$.
- **ADDATEND(e)**: Increase the size n of D by one. Then set $D[n-1] := e$.

- `DELETEATFRONT()`: Shift all elements one position to the left, the old value of $D[0]$ will be lost. Then decrease the size n by one.
- `DELETEATEND()`: Shift all elements one position to the right, the old value of $D[n - 1]$ will be lost. Then decrease the size n by one.

Interestingly, the specification of the ADT `DEQUE` C++ standard library (STL) requires all these operations to be performed in $O(1)$ time. However, as it is remarked in [Mor 01], none of the mainstream implementations fullfills these requirements.

A deque of bounded size $n \leq N$ can be implemented as a **circular array**, that is an ordinary array of capacity N together with an offset p to the first element. The location of the i th element then is $A[p + i \bmod N]$.

If n exceeds the bound N , we allocate a new array of capacity $2N$ and move all the elements from the old to the new array. If an element is removed and n becomes smaller than $N/3$, we move all elements to a new array of size $N/2$. This is the well-known **doubling technique**. Though there is nothing magic with the values $N/2$ and $N/3$ values have to be choosen in a way that yields $O(1)$ amortized time for all deque-operations. Here for each time the capacity changes it takes $\Omega(n)$ time until it will change again. The doubling technique has its obvious disadvantage in online algorithms, interactive environments, or distributed computing. In all three cases we wish to have constant time bounds for each single operation since we do not want parts of the system to wait for another part in order to finish a basic task as extending an array. In such a case, **deamortization** helps. A deamortization technique which can be employed in many cases works as follows: Instead of one single array A of size N we keep a second array B of size $2N$, where now we have $N \leq n < 2N$. From the n elements of the deque, we store the first $2N - n$ in A and the last $2(n - N)$ elements in B . Adding an element at the front implies to move two elements from A to B while adding at the end of the deque implies to move one element from B to A . By the reversed process we can delete elements a the front or at the end.

Now if B overflows ($n > 2N$), A is empty and we assign B to A and allocate a new array for B of size $4N$. If B becomes empty ($n < N$), we assign A to B and allocate a new array for A of size $N/2$.

This deamortization needs extra memory for up to $2n$ elements. In some unfortunate cases of periodically adding and removing a single element, it may happen that we release and allocate new memory for each single operation. Literally, we are going back and forth on our heels. We can avoid this at the expense of even more extra memory by buffering the last released array until n achieves a certain limit. The doubling technique and its deamortization described above work with many other datastructures, for instance hashtables. In our special case of deques, we can save most of the extra memory if the entries in A and B are not the elements themselves but pointers to **buckets** of elements, i.e. arrays of a fixed size k . This extra level of indirection almost doubles the adresssing time for random access. On the other hand, for typical bucket sizes $k = 512, 1024, 2048, \dots$ iteration time is still almost the same and the extra memory for $3n/k$ elements is neglectible. As an advantage we do not need to move the elements around but only the pointers to the buckets — their number is far less than the number of elements. The article [BCDMS 99] shows a more elaborate way with growing bucket size k in order to achieve

the tight approximative bound of $\Theta(\sqrt{n})$ extra memory. Their method has a slightly more complicated addressing scheme but we can achieve the same result by toggling between two different modes: In the first mode, A and B both have the same number of buckets, but the buckets in B 's are twice as large as in A . In the second mode, both bucket sizes are the same but B has twice as many buckets as A has. Note that either way B can hold exactly twice as many elements as A . The mode changes always when addition (deletion) of an element requires the creation of a new B (A).

Now, when implementing very fast extendible arrays — without using buckets as an extra-level of indirection — it is quite disturbing that we need to move two elements when adding or deleting an element at the front of the deque. We introduce a slightly better scheme: We split the deque with $n \leq 2$ elements into three parts D_1 , D_2 , and D_3 . N is chosen as a power of 2 such that $N < n \leq 2N$, and we create two ringbuffers: The first one, A , is of size N , the second ringbuffer B is of size $2N$. The middle part D_2 of the deque has exactly $N - (n - N) = 2N - n$ elements, which always fits into the smaller ringbuffer A . There are $n - (2N - n) = 2(n - N)$ elements left, which is always an even number. So we can arrange the splitting in such a ways that D_1 and D_3 have exactly the same number of elements, and arrange them in the ringbuffer B so that D_1 starts and ends exactly at the opposite site where D_3 starts and ends. The actual start and end inside the ringbuffer are, of course, subject to dynamical change.

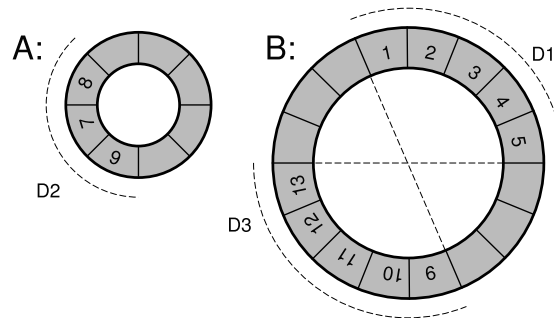


Figure B.1: The deque D_1, D_2, D_3 in the arrays A and B

Important is that D_1 and D_3 have always the same size, and if D_2 is empty, they meet each other at both ends and fill the ringbuffer B . If B is full, it becomes the new A and a B is being assigned a new ringbuffer size $4N$ is created. Similar if B becomes empty so that A is full, then A becomes the new B and a new ringbuffer of size $N/2$ (except for $n = 1$) is created. Now, if we add an element at the front of our deque, we add it at the front of D_1 and move an element from the end of D_2 to the front of D_3 . Deleting an element from the deque implies moving another element from the front of D_3 to the end of D_2 . Adding and deleting at the end of the deque is done in a similar way.

Bibliography

- [AHU 74] A.V. Aho, J.E. Hopcroft, J.D. Ullman *The Design and Analysis of Algorithms*. Addison-Wesley, 1974
- [AHT 00] S. Alstrup, J. Holm, M. Thorup *Maintaining Center and Median in Dynamic Trees*. Proc. 7th SWAT, LNCS 1851, 2000
- [AHLT 03] S. Alstrup, J. Holm, K. de Lichtenberg, M. Thorup *Maintaining Information in Fully-Dynamic Trees*. The Computing Research Repository (CoRR) cs-DS-0310065, 2003
- [Bar 04] N. Barraci *Persistente Arrays zur Effizienten Speicherung sich Partiell Wiederholender Objektketten*. Diploma Thesis, Darmstadt University of Technology, 2004
- [BCDMS 99] A. Brodnik, S. Carlsson, E.D. Demaine, J.I. Munro, R. Sedgewick *Resizable Arrays in Optimal Time and Space*. 6th International Workshop on Algorithms and Data Structures (WADS) 1999
- [BHKS] W. Brauer, M. Holzer, B. König, S. Schwoon *The Theory of Finite State Adventures*. EATCS Bulletin, 79:230-237, 2003
- [BJM 98] C. Barret, R. Jacob, M. Marathe *Formal Language Constrained Path Problems*. Los Alamos Reserach Report LA-UR-98-1739, 1998
- [Bur 02] J. Burghardt *Maintaining Partial Sums In Logarithmic Time*. Nordic Journal of Computation, 2002
- [CeDi 04] M. Cetti, S. Dingel *Solution to an Exercise*. private communication, 2004
- [CoWi 90] D. Coppersmith, S. Winograd *Matrix Multiplication Via Arithmetic Progression*. Journal of Symbolic Computation 9, 1990
- [CLRS 01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein *Introduction to Algorithms*. The MIT Press, 2001, second edition
- [EGI 99] D. Eppstein, Z. Galil, G.F. Italiano *Dynamic Graph Algorithms*. in *Algorithms and Theory of Computation Handbook* CRC Press, 1999

- [Fre 85] G.N. Fredrickson *Data Structures for Online Updating of Minimum Spanning Trees*. SIAM Journal of Computing 14, 1985
- [GaJo 79] M.R. Garey, D.S. Johnson *Computers and Intractability — A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979
- [Gli 03] O. Glier *Kolmogorov Complexity and Deterministic Context-Free Languages*. SIAM Journal of Computing 32, 2003
- [Gon 95] L. Gonick *The Solution*. Discover, April 1995
- [Goo 98] J. T. Goodman *Parsing Inside-Out*. Thesis, Havard University, 1998
- [Har 78] M.A. Harrison *Introduction to Formal Language Theory*. Addison-Wesley 1978
- [Has 98] R. Hassin *Approximation schemes for the restricted shortest path problem*. Mathematics of Operations Research, 17(1), 1992
- [HeKi 99] M.R. Henzinger, V. King *Randomized Fully Dynamic Graph Algorithms with Poly-Logarithmic Time per Operation*. Journal of the ACM 46, 1999
- [HLT 01] J. Holm, K. de Lichtenberg, M. Thorup *Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-edge, and Biconnectivity*. Journal of the ACM 48, 2001
- [Knu 73] D.E. Knuth *The Art of Computer Programming Vol. 3, Sorting and Searching*. Addison-Wesley 1978
- [KnPl 81] D.E. Knuth, M.F. Plass *Breaking Paragraphs Into Lines*. Software: Practice & Experience, 11(11), 1981
- [Knu 77] D.E. Knuth *A Generalization of Dijkstra's Algorithm*. Information Processing Letters 6, 1977
- [Knu 98] D.E. Knuth *The Art of Computer Programming Vol. 2, Seminumerical Algorithms*. Addison-Wesley 1998, third edition
- [KuSa 85] W. Kuich, A. Salomaa *Semirings, Automata, Languages*. Springer-Verlag 1985
- [Lee 02] L. Lee *Fast Context-Free Parsing Requires Fast Boolean Matrix Multiplication*. Journal of the ACM 49(1), 2002
- [Moh 02] M. Mohri *Semiring Frameworks and Algorithms For Shortest-Distance Problems*. Journal of Automata, Languages and Combinatorics 7, 2002
- [MoRa 95] R. Motwani, P. Raghavan *Randomized Algorithms*. Cambridge University Press, 1995

- [Mor 01] B.B. Mortensen *The Deque Class in the Copenhagen STL: First Attempt*. CPH STL Report 2001-4, 2001
- [Oka 98] C. Okasaki *Purely Functional Data Structures*. Cambridge University Press 1998
- [Ren 02] K. Renz *Algorithms and Data Structures for a Music Notation System based on GUIDO Music Notation*. Phd thesis, Technische Universität Darmstadt 2002
- [Ryt 95] W. Rytter *Context-Free Recognition via Shortest Paths Computation: A Version of Valiant's Algorithm*. Theoretical Computer Science 143, 1995
- [Sei 95] R. Seidel *On the All-Pairs-Shortest-Paths Problem in Unweighted Undirected Graphs*. Journal of Computing and System Sciences 51, 1995
- [Val 74] L.G. Valiant *General Context-Free Recognition in Less Than Cubic Time*. Journal of Computing and System Sciences 10, 1974
- [Wal 84] H.K.-G. Walter *A Simple Proof of Valiant's Lemma*. Theoretical Informatics and Applications 20, 1986
- [LiVi95] Ming Li and Paul Vitányi. *A New Approach to Formal Language Theory by Kolmogorov Complexity*. SIAM J. Comput., 24:2, 1995
- [LiVi97] Ming Li and Paul Vitányi *An Introduction to Kolmogorov Complexity and Its Applications*. Second Edition Springer-Verlag, 1997

Oliver Glier – Curriculum vitae

16th January 1972, Frankfurt am Main, Germany

Ziegelhüttenweg 86
60598 Frankfurt am Main
Germany

Career History

- | | |
|-------------|--|
| 1999 - 2004 | research assistant at the research group for Automata Theory and Formal Languages (AFS), Department of Computer Science, Darmstadt University of Technology, Germany |
| 2001 - 2002 | one-year visit at Fakultas Ilmu Komputer, Universitas Indonesia in Depok/Jakarta, Indonesia. DAAD scholarship |
| 1999 | diploma in computer science with a minor in mathematics |
| 1992 - 1999 | student of computer science and mathematics at Darmstadt University of Technology, Germany. |
| 1991 - 1992 | community service at the Day Nursery Gartenstraße, Neu-Isenburg, Germany |
| 1978 - 1991 | elementary and grammar school, 1991 Abitur |

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit zur Erlangung des akademischen Grades Dr.rer.nat. mit dem Titel “Persistent Arrays, Path Problems, and Context-free Languages” selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keine Promotionsversuche unternommen.

Darmstadt, den

Oliver Glier